SQL Fundamentals MySQL (Modulul 1)

Cuprins

1. Introducere despre bazele de date

1.1 Introducere

Aplicaţiile informatice utilizate în prezent lucrează cu un număr foarte mare de date care trebuie stocate în aşa fel încât să le putem accesa rapid şi uşor. Astfel, majoritatea aplicaţiilor, de la site-uri şi alte aplicaţii web până la aplicaţii bancare sau de gestiune a clienţilor, folosesc baze *de date relaţionale*.

În acest curs de **Fundamente Baze de date MySQL** ne propunem să parcurgem câteva noţiuni teoretice fundamentale pentru înţelegerea conceptelor folosite în lucrul cu baze de date relaţionale, concepte tot mai des întâlnite în limbajul informatic curent (*informaţie*, *date*, *baze de date*, etc.) dar şi să trecem în revistă etapele care sunt parcurse în realizarea aplicaţiilor 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 experienţă şi cunoştinţe în domeniul bazelor de date, dar şi persoanelor care au cunoştinţe şi o minimă experienţă în lucrul cu baze de date.

Noţiunile teoretice fundamentale despre bazele de date relaţionale, precum şi elementele limbajului de interogare **SQL**, sunt introduse pas cu pas şi sunt însoţite de exemplificări şi utilizări practice.

Am ales ca mediu de dezvoltare a aplicaţiilor 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 aplicaţiile şi site-urile web împreună cu limbajul de programare PHP.

1.1.1Aplicatii ale bazelor de date pe Web

Orice site care are un modul de creare cont şi login, orice magazin online, site de ştiri, blog, etc. are o bază de date în care este ţinută informaţia 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 doreşte să lucreze în domeniul programării web, pe lângă cunoştinţele de HTML şi CSS folosite în partea de dezvoltare a aplicaţiei ce interacţionează cu utilizatorul, şi cele de programare PHP, are nevoie şi de cunoştinţe de baze de date relaţionale, întrucât aproape toate site-urile şi toate aplicaţiile web conţin informaţia în baze de date relaţionale. Jobul de programator web este unul foarte interesant şi există o cerere mare pe piaţă pentru persoane care au cunoştinţe de programare web. Este un job provocator, pentru că fiecare proiect aduce ceva nou, la fiecare proiect se pot învăţa lucruri suplimentare şi aduce şi satisfacţia că produsul realizat este vizualizat de foarte mulţi oameni (ne referim aici la site-urile web).

Tipologia site-urilor web porneşte de la site-uri de prezentare şi 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 informaţia introdusă dintr-o secţiune de administrare, în acest sens folosindu-se un limbaj de programare şi baze de date relaţionale.

Iată modul de funcţionare şi de interacţiune cu serverul web şi 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 fişierele ce conţin aceste pagini au extensia .html sau .htm nu avem nevoie să instalăm altceva pe calculatorul nostru pentru a putea deschide acele pagini.

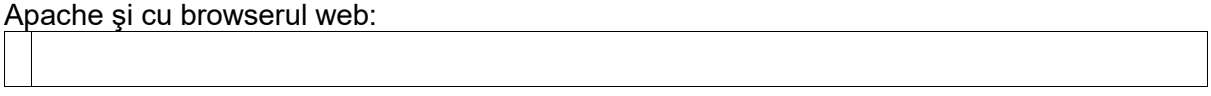
În schimb, dacă vom realiza un site web dinamic folosind limabjul PHP şi baze de date **MySQL**, avem nevoie de instalarea pe calculator a unui server.

PHP este un limbaj care funcţionează 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 şi generează cod HTML pe care îl transmite către browser astfel că browserul web primeşte tot cod HTML, singurul pe care ştie să îl interpreteze şi să îl afişeze.

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

În schema de mai jos vom reprezenta modul în care interacţionează PHP-ul cu serverul Apache şi 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.
Severul trimite pagina spre procesare interpretorului PHP (2).

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

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

Serverul Apache returnează clientului cod HTML, pe care acesta ştie să îl interpreteze şi să îl afişeze (6).

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

1.1.2 Aplicaţii ale bazelor de date în alte domenii

O mare parte dintre aplicaţiile software existente folosesc baze de date pentru stocarea şi extragerea informaţiilor. Aceste baze de date se interoghează ulterior pentru obţinerea de diverse statistici. De exemplu, informaţiile despre clienţii şi facturile unei companii se ţin într-o bază de date.

1.2 Obiective

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

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

- să înţelegeţi noţiunile teoretice folosite în domeniul bazelor de date (dată, informaţie, bază de date, sistem de gestiune a bazelor de date, tabelă, atribut, înregistrare, etc.);
- să înţelegeţi principiile generale ale algebrei relaţionale, să cunoaşteţi operaţiile de reuniune, intersecţie, diferenţă, produs cartezian folosite în teoria mulţimilor;
- să înţelegeţi principiul normalizării bazelor de date
- să cunoaşteţi şi să puteţi instala mediile de lucru folosite pentru lucrul cu baze de date **MySQL** (serverul **WAMP** şi **HeidiSQ/MySQL Workbench**);
- să înţelegeţi noţiunea de relaţie între tabele şi să cunoaşteţi tipurile de relaţii între tabele;
- să puteţi proiecta o bază de date;
- să cunoaşteţi sintaxa **SQL** şi instrucţiunile de bază (**INSERT**, **UPDATE**, **DELETE**, **SELECT**);
- să cunoaşteţi limbajul de descriere a datelor (**LDD**), limbajul de manipulare a datelor (**LMD**), precum şi **tipuri de date** şi **operatori MySQL**;
- să cunoaşteţi câteva **funcţii predefinite MySQL** (pentru lucrul cu şiruri de caractere, cu date calendaristice, funcţii matematice, etc.);
- să înţelegeţi noţiunile de **join**, **union** şi **subinterogări** şi să le folosiţi în interogări complexe;
- să înţelegeţi utilizarea **vederilor** (**tabele virtuale**) şi 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 **inserate 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 puteţi realiza aplicaţii 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 obţine diploma finală, înainte de a susţine testul final ce va conţine întrebări de tip grilă, trebuie să prezentaţi o aplicaţie complexă în care să fie folosite noţiunile învăţate. Este vorba despre o bază de date care asigură gestiunea unui anumit domeniu (de exemplu o bază de date pentru gestiunea cărţilor dintr-o bibliotecă sau o bază de date folosită la gestiunea angajaţilor unei companii, o bază de date folosită la gestiunea pacienţilor şi medicilor dintr-un spital, etc.).

1.3 Prezentare lecţii

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

În prima lecție a cursului sunt prezentate pe larg **noţiuni teoretice** precum şi câteva noţiuni fundamentale de **algebră relaţională**. Înţelegerea conceptului de normalizare precum şi prezentarea formelor normale reprezintă următorul subiect abordat în partea de început.

Cea de-a doua lecţie se axează pe prezentarea concretă a mediului de lucru (**HeidiSQL/MySQL Workbench**) şi pe prezentarea modului în care este proiectată o bază de date. De asemenea, vom aborda subiectul relaţionării – concept fundamental în bazele de date relaţionale – şi vom explica şi exemplifica tipurile de relaţii ce pot exista între tabelele unei baze de date. Continuăm această lecţie cu o introducere în limbajul **SQL** (**S**tructured **Q**uery **L**anguage).

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

În cea de-a **patra lecţie** prezentăm **L**imbajul de **M**anipulare a **D**atelor (**LMD**), în limba engleză **D**ata **M**anipulation **L**anguage (**DML**). Limbajul de Manipulare a Datelor se referă la cele patru instrucţiuni fundamentale ale limbajului **SQL**:

- **INSERT** – pentru introducerea înregistrărilor într-o tabelă;
- **UPDATE** – pentru modificarea înregistrărilor din tabele;
- **DELETE** – pentru ştergerea înregistrărilor din tabele;
- **SELECT** – instrucţiunea de regăsire a informaţiilor din baza de date.

Lecţia următoare (**lecția 5**) prezintă noţiuni despre **operatorii** şi **funcţiile predefinite** ale **MySQL**. Este vorba de operatorii aritmetici, logici şi de comparare şi de funcţii matematice, funcţii pentru lucrul cu şiruri de caractere, cu date calendaristice, etc.

Cursul continuă, în **lecția 6**, cu noţiuni teoretice despre **JOIN**-uri, tipuri de **JOIN**-uri şi **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** şi **tipurile de subinterogări**.

Penultima lecție (**lecția 8**) este dedicată prezentării **tabelelor virtuale** (**view**-uri sau **vederi**) și **tabelelor 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 apot fi aplicați pe coloanele unei tabele dintr-o bază de date.

Ultima lecție conține o referire amănunțită asupra **conceputului 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 Noţiuni teoretice

1.4.1Date, informaţii, cunoştinţe

Auzim adesea folosindu-se termenii „societate informaţională”, „tehnologia informaţiei” însă de multe ori cuvântul „informaţie” este folosit fără a înţelege clar sensul acestui cuvânt şi faptul că este o diferenţă între **date**, **informaţii**, **cunoştinţe** şi **obiecte**.

În general, conţinutul gândirii umane operează cu următoarele concepte:
Date – constau în material brut, fapte, simboluri, numere, cuvinte, poze fără un înţeles de sine stătător, neintegrate într-un context, fără relaţii cu alte date sau obiecte. Ele se pot obţine în urma unor experimente, sondaje etc.

Informaţii – prin prelucrarea datelor şi găsirea relaţiilor dintre acestea se obţin informaţii care au un înţeles şi sunt integrate într-un context. Datele organizate şi prezentate într-un mod sistematic pentru a sublinia sensul acestor date devin informaţii. Pe scurt *informaţiile sunt date prelucrate*. Informaţiile se prezintă sub formă de rapoarte, statistici, diagrame etc.

Cunoştinţele sunt colecţii de date, informaţii, adevăruri şi principii învăţate, acumulate de-a lungul timpului. Informaţiile despre un subiect reţinute şi înţelese şi care pot fi folosite în luarea de decizii devin cunoştinţe. Cu alte cuvinte, cunoştinţele apar în momentul utilizării informaţiei.

Obiectele reprezintă cunoştinţe pentru care se ştie comportamentul şi proprietăţile, referitoare la o entitate din lumea reală.

1.4.2Bază de date

O *bază de date* (**BD**) conţine toate informaţiile necesare despre obiectele ce intervin într-o mulţime de aplicaţii, relaţiile logice între aceste informaţii şi tehnicile de prelucrare corespunzătoare.

O *bază de date* reprezintă o colecţie de date organizate ce pot fi accesate simultan de mai mulţi utilizatori. Prelucrarea datelor se referă la operaţiile de *introducere*, *ştergere*, *actualizare* şi *interogare* a datelor. O *bază de date* este o colecţie de înregistrări sau de informaţii introduse şi 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**) şi răspunde cu informaţie, în funcţie de care se iau decizii. Pentru valorificarea informaţiei ce poate fi extrasă dintr-o bază de date, este esenţial modul în care organizăm şi stocăm datele într-o bază de date.

1.4.3Sistem 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 informaţiilor în bazele de date şi dezvoltarea de aplicaţii privind bazele de date.

Cu alte cuvinte, aplicaţia care este folosită pentru a realiza, a administra şi 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 interacţiunea utilizatorului cu baza de date. Orice **SGBD** conţine, printre alte componente un limbaj de descriere a datelor (**LDD**) care permite descrierea structurii bazei de date, a fiecărei componente a ei, a relaţiilor dintre componente, a drepturilor de acces ale utilizatorilor la baza de date, a restricţiilor în reprezentarea informaţiilor.

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

Teoria relaţională, foarte bine pusă la punct într-un domeniu de cercetare distinct, a dat o fundamentare solidă realizării de **SGBD**-uri performante. La sfârşitul anilor ‘80 şi apoi în anii ‘90 au apărut, în special o dată cu pătrunderea în masă a microcalculatoarelor, numeroase sisteme de gestiune bazelor de date relaţionale (**SGBDR**-uri).

Aceasta a însemnat o evoluţie de la **SGBD**-urile de generaţia întâi (arborescente şi reţea) spre cele de generaţia a doua (relaţionale). Această evoluţie s-a materializat, în principal în: oferirea de limbaje de interogare neprocedurale, îmbunătăţirea integrităţii şi securităţii datelor, optimizarea şi simplificarea acceselor.

Teoria relaţională este un ansamblu de concepte, metode şi instrumente care a dat o fundamentare riguroasă realizării de **SGBDR** performante.
SGBD relaţional este un ansamblu de produse software complex şi complet, care implementează modelul logic de date relaţional, precum şi cel puţin un limbaj de programare relaţional.
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 cerinţele maximale pentru ca un **SGBD** să fie relaţional.

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

R_0. *Gestionarea datelor se face la nivel de relaţie*: limbajele utilizate trebuie să lucreze cu relaţii (tabele). Relaţia trebuie să fie unitatea de informaţie pentru operaţii.
R_1. *Reprezentarea logică a datelor*: toate informaţiile din **BDR** trebuie stocate şi prelucrate ca relaţii (tabele).
R_2. *Garantarea accesului la date*: **LMD** trebuie să permită accesul la fiecare valoare atomică din **BDR** (tabelă, coloană, cheile de diferite tipuri).
R_3. *Valoarea NULL*: trebuie să se permită mai întâi declararea şi 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 alţi operatori (de exemplu, operatorul existenţial IS şi operatorul logic NOT).
R_4. *Metadatele*: informaţiile despre descrierea **BDR** se stochează în dicţionar ca tabele, la fel ca datele propiu-zise.
R_5. *Limbajele utilizate*: **SGBDR** trebuie să permită utilizarea mai multor limbaje, dintre care cel puţin unul să permită definirea tabelelor (de bază şi virtuale), definirea restricţiilor de integritate (constrângeri), manipularea datelor, autorizarea accesului, tratarea tranzacţiilor.
R_6. *Actualizarea tabelelor virtuale*: trebuie să se permită ca tabelele virtuale (view-uri) să fie şi efectiv actualizabile, nu numai teoretic actualizabile.
R_7. *Actualizările în baza de date*: manipularea unei tabele trebuie să se facă prin operaţii de regăsire (interogare) dar şi de actualizare.
R_8. *Independenţa fizică a datelor*: schimbarea stucturii fizice a datelor (modul de reprezentare (organizare) şi modul de acces) nu afectează programele.
R_9. *Independenţa logică a datelor*: schimbarea structurii de date (logice) a tabelelor nu afectează programele.
R_{10}. *Restricţiile de integritate*: acestea trebuie să fie definite prin **LDD** şi stocate în dicţionarul (catalogul) **BDR**.
R_{11}. *Distribuirea geografică a datelor*: **LMD** trebuie să permită ca programele de aplicaţie să fie aceleaşi atât pentru date distribuite cât şi petru date centralizate (alocarea şi localizarea datelor vor fi în sarcina **SGBDR**-ului).
R_{12}. *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 restricţiile de integritate.

Regulile lui Codd sunt greu de indeplinit în totalitate de către **SGBDR**. Pornind de la cele 13 reguli de mai sus, au fost formulate o serie de criterii (cerinţe) pe care trebuie să le îndeplinească un **SGBD** pentru a putea fi considerat relaţional într-un anumit grad. S-a ajuns astfel, la mai multe grade de relaţional pentru **SGBDR**: cu interfaţă relaţională (toate datele se reprezintă în tabele, există operatorii de selecţie, proiecţie şi joncţiune doar pentru interogare), pseudorelaţional (toate datele se reprezintă în tabele, există operatorii de selecţie, proiecţie şi joncţiune fără limitări), minimal relaţional (este pseudorelaţional şi în plus, operaţiile cu tabele nu fac apel la pointeri observabili de utilizatori), complet relaţional (este minimal relaţional şi

în plus, există operatorii de reuniune, intersecţie şi diferenţă, precum şi restricţiile de integritate privind unicitatea cheii şi restricţia referenţială).

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

- reguli de bază (fundamentale): R_0 şi R_{12};
- reguli structurale: R_1 şi R_6;
- reguli pentru integritatea datelor: R_3 şi R_{10};
- reguli pentru manipularea datelor: R_2, R_4, R_5 şi R_7;
- reguli pentru independenţa datelor: R_8, R_9 şi R_{11}.

În concluzie, **SGBDR** este un sistem software complet care implementeză modelul de date relaţional şi respectă cerinţele impuse de acest model. El este o interfaţă între utilizatori şi baza de date.

1.4.5 Exemple de SGBD-uri

Oracle. Este realizat de firma Oracle Corporation USA. Sistemul este complet relaţional, robust, se bazează pe **SQL** standard extins. Arhitectura sistemului este client/server, permţând lucrul, cu obiecte şi distribuit. Sistemul respectă teoria relaţională, este robust şi se bazează pe **SQL** standard. Permite lucrul distribuit şi 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** şi 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 şi distribuit sub Licenţa Publică Generală **GNU** (în engleză GNU General Public License, prescurtat **GNU GPL** – este o licenţă software care are scopul de a da dreptul oricărui utilizator de a copia, modifica şi redistribui programe şi coduri sursă ae programatorilor care îşi licenţiază 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 propiu foarte puternic, o extensie orientată obiect, programare vizuală şi nucleu extins de **SQL**.

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

Modelul unei baze de date este o specificaţie tehnică acceptată de mai mulţi furnizori de programe de baze de date (**DBMS**) ce se referă la modul în care sunt stocate informaţiile în baza de date şi modul în care sunt folosite.

Exemple de modele sunt: **modelul relaţional**, **modelul orientat-obiect**, **modelul ierarhic**, etc.
Cel mai răspândit în prezent este **modelul relaţional**. Bazele de date relaţionale au informaţiile organizate în tabele, iar între informaţiile din aceste tabele pot fi stabilite legături.

1.4.6 Colectarea şi analizarea datelor. Modelul conceptual

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

Modelul conceptual al datelor este o reprezentare vizuală clară şi corectă a activităţii unei organizaţii.

Modelul conceptual al datelor include entităţile (informaţiile) majore şi relaţiile dintre acestea şi nu conţine informaţii detaliate privind atributele (caracteristicile) entităţilor. Este generat prin identificarea cerinţelor afacerii modelate, aşa cum rezultă din documente, discuţii cu personalul implicat, cu analişti şi experţi ai domeniului studiat, cu beneficiarii finali. Modelul conceptual realizat va fi prezentat echipelor funcţionale în vederea revizuirii.
În această etapă sunt analizate natura şi modul de utilizare a datelor. Sunt identificate datele care vor trebui memorate şi procesate, se împart aceste date în grupuri logice şi se identifică relaţiile 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 şi a modului de utilizare a acestora, vom realiza o bază de date care putem constata în final că nu întruneşte cerinţele beneficiarului. Costurile modificării acestei baze de date sunt mult mai mari decât costurile pe care le-ar fi implicat etapa de analiză şi realizare a modelului conceptual. Modificarea modelului conceptual este mult mai uşoară decât modificarea unor tabele deja existente, care eventual conţin şi o mulţime de date. Informaţiile obţinute în etapa documentării vor fi reprezentate într-o formă convenţională care să poată fi uşor înţeleasă de toată lumea.

O astfel de reprezentare este diagrama entităţi-relaţii numită şi harta relaţiilor **ERD** (**E**ntity **R**elationship **D**iagram). Aceste scheme sunt un instrument util care uşurează comunicarea dintre specialiştii care proiectează bazele de date şi programatori, pe de o parte, şi beneficiari, pe de altă parte. Aceştia din urmă pot înţelege cu uşurinţă o astfel de schemă, chiar dacă nu sunt cunoscători în domeniul IT. Diagramele **ERD** sunt uşor de creat şi de actualizat, însă, avantajul major al lor este dat de simplitatea reprezentărilor ce facilitează înţelegerea şi de către nespecialişti.

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 înţelegerea de către utilizatori a datelor şi a relaţiilor dintre acestea;
- sunt independente de implementare.

Iată în continuare principalele elemente care intră în componenţa unui **ERD** precum şi convenţiile de reprezentare a acestora.

1.4.7 Entităţi. Instanţe. Atribute. Identificator unic.

O entitate este un lucru, obiect, persoană sau eveniment care are semnificaţie pentru afacerea modelată, despre care trebuie să colectăm şi 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 operaţie, sau

poate fi o noţiune 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 colţurile rotunjite. Numele entităţii este întotdeauna un substantiv la singular şi se scrie în partea de sus a dreptunghiului cu majuscule, ca în figura de mai jos:

Fig.1.1

Entităţile sunt clase de obiecte de acelaşi tip, un obiect al clasei reprezentând o instanţă a entităţii.

O instanţă a unei entităţi 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 instanţă a entităţii ELEV.

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

Aşadar, după ce am identificat entităţile trebuie să descriem aceste entităţi în termeni reali, adică să le stabilim atributele.

Entităţile 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), şir de caractere (ex. Popescu, XII A), dată calendaristică (ex. 12/02/2010) etc.

Un *atribut* este orice detaliu care serveşte la identificarea, clasificarea, cuantificarea, sau exprimarea stării unei instanţe a unei entităţi. Atributele sunt informaţii specifice ce trebuie cunoscute şi memorate.
În cadrul unui **ERD**, atributele se vor scrie imediat sub numele entităţii, cu litere mici. Un *atribut* este un substantiv la singular.

Atributele cărora le poate lipsi valoarea se numesc *atribute opţionale*, iar cele cărora trebuie să le atribuim o valoare se numesc *atribute obligatorii*. Dacă un atribut este obligatoriu, pentru fiecare instanţă a entităţii respective trebuie să avem o valoare pentru acel atribut, de exemplu este obligatoriu să cunoaştem numele elevilor.

Pentru un atribut opţional putem avea instanţe pentru care nu cunoaştem valoarea atributului respectiv. De exemplu atributul email al entităţii ELEV este opţional, un elev putând să nu aibă adresă de email. Un atribut obligatoriu este precedat în **ERD** de un asterisc „*”, iar un atribut opţional va fi precedat de un cerculeţ „o”.

Fig. 1.2

De exemplu atributele entităţii ELEV sunt nume, prenume, adresa, număr de telefon, adresa de email, data naşterii etc.

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

De exemplu combinaţia dintre titlu, numele autorului şi data apariţiei poate forma unicul identificator al entităţii CARTE. Oare combinaţia titlu şi 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 entităţi vor fi precedate de semnul diez „#”. Atributele din **UID** sunt întotdeauna obligatorii, însă semnul „#” este suficient, nu mai trebuie pus şi un semn asterisc în faţa 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 naşterii) 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 instanţă a entităţii (valoarea sau combinaţia de valori ale atributelor este unică pentru fiecare instanţă) se numeşte **identificator unic** (**Unique IDentifier** - **UID**).

Identificatorii unici sunt obligatorii şi sunt precedaţi de caracterul # (diez). Se pot folosi identificatori unici *naturali* (atribute ce au o semnificaţie concretă pentru entitatea respectivă) sau *artificiali* (atribute create şi menţinute 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 entităţii este *id_candidat*, un **UID** de tip artificial. Un **UID** natural al entităţii poate fi stabilit atributul *cnp*, întrucât nu există două persoane cu acelaşi cod numeric personal, sau o pereche de atribute ce ar asigura unicitatea: perechea (nume, iniţiala, prenume, adresa).
În continuare aveţi reprezentarea grafică a entităţii CANDIDAT:

Fig 1.4 Entitatea CANDIDAT

Atributele obligatorii sunt nume, initiala, prenume, cnp, mediul, iar cele opţionale serii_anterioare, taxa, adresa, telefon, data_inscriere. Se recomandă evitarea folosirii atributelor ale căror valori se modifică frecvent (de exemplu vârsta unei persoane), aşa-numitele atribute volatile. Acestea pot fi înlocuite cu atribute non-volatile (de ex. data naşterii, 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 naşterii, sex, vârstă, întrucât ele pot fi obţinute prin prelucrări elementare din atributul cnp.

1.4.8 Etapele realizării unei aplicaţii informatice

Realizarea unei aplicaţii informatice ce utilizează baze de date necesită parcurgerea unei succesiuni de etape: analiza sistemului, proiectarea bazei de date, realizarea componentelor logice, punerea în funcţiune, dezvoltarea şi întreţinerea (mentenanţa).

1.4.8.1 Analiza sistemului informatizat

Scopul analizei sistemului este de a evidenţia cerinţele aplicaţiei şi resursele utilizate, precum şi de a evalua aceste cerinţe prin modelare (analiza). Studiul situaţiei existente se realizează prin: identificarea caracteristicilor generale ale unităţii, identificarea activităţilor desfăşurate, identificarea resurselor existente (informaţionale, umane, echipamente), identificarea necesităţilor de prelucrare.

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

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

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

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

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

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

Rezultatul analizei dinamice este *modelul dinamic*.

c) *Analiza funcţională* evidenţiază modul de asigurare a cerinţelor informaţionale (fluxul prelucrărilor) din cadrul sistemului, prin care intrările sunt transformate în ieşiri. Prin analiza funcţională se delimitează:

- Aria de cuprindere a aplicaţiei informatice;
- Se identifică sursele de date;
- Se identifică modul de circulaţie şi prelucrare a datelor;
- Se identifică apoi rezultatele obţinute.

Rezultatul analizei funcţionale este *modelul functional*

1.4.9 Modelul relațional

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

Modelul relaţional este definit prin: structura de date, operatorii care acţionează asupra structurii şi restricţiile de integritate.
Conceptele utilizate pentru definirea structurii de date
sunt: domeniul, tabela (relaţia), atributul, tuplul, cheia şi schema tabelei.

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/relaţia este un subansamblu al produsului cartezian al mai multor domenii, caracterizat printr-un nume, prin care se definesc atributele ce aparţin aceleaşi clase de entităţi. 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* reperezintă 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* şi nu are nume. Ordinea liniilor (tupluri) şi coloanelor (atribute) dintr-o tabelă nu trebuie să prezinte nici o importanţă.

Schema tabelei este formată din numele tabelei, 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 evidenţă şi legăturile dintre tabele. Definirea legăturilor dintre tabele sau a relaţiilor dintre tabele se face logic construind asocieri între tabele cu ajutorul unor atribute de legătură.

Cardinalitatea relaţiei este egală cu numărul de linii sau tupluri conţinute de un tabel.

Cheia primară a unei tabele 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 numeşte *cheie compusă*. *Orice tabelă are o cheie primară care trebuie să aibă valori unice şi nenule*.

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

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

Un *atribut* ce îndeplineşte condiţiile necesare cheii primare se numeşte *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 iniţială se numeşte *cheie externă* iar cel din tabela finală este *cheie primară*.

Cheia externă (foreign key) este atributul sau ansamblul de atribute ce serveşte 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.

Condiţiile pe care trebuie să le îndeplinească cheile externe sunt următoarele:

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

Această ultimă condiţie este cea a *integrităţii referinţei*, 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 conţinute 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 forţate să fie doar dintre cele ale cheii primare corespondente. Aceasta poartă numele de *constrângere de integritate referenţială*.

Constrângerile de integritate sunt verificate automat de **SGBD** atunci când au loc operaţii de modificare a conţinutului unei tabele (introducere, modificare sau ştergere de înregistrări). În cazul în care valorile nu sunt valide operaţia nu se efectuează şi 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** – defineşte cheia primară;
- **UNIQUE** – defineşte uncitatea;
- **FOREIGN KEY** – defineşte o cheie externă;
- **CHECK** – introduce o condiţie (expresie logică).

Avem următoarele legături posibile între tabele:

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

Potenţial, orice tabelă se poate lega cu orice tabelă, după orice atribute. Legăturile se stabilesc la momentul descrierii datelor prin limbaje de descriere a datelor (**LDD**), cu ajutorul restricţiilor de integritate. Practic, se stabilesc şi legături dinamice la momentul execuţiei.

1.4.10 Operatorii modelului relaţional

Operatorii modelului relaţional sunt operatorii din *algebra relaţională* şi operatorii din *calculul relaţional*.

Algebra relaţională este o colecţie de operaţii formale aplicate asupra tabelelor (relaţiilor), şi a fost concepută de E.F.Codd. Operaţiile sunt aplicate în expresiile algebrice relaţionale care sunt cereri de regăsire. Acestea sunt compuse din operatorii relaţionali şi operanzi.

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

Algebra relaţională are cel puţin puterea de regăsire a calculului relaţional. O expresie din calculul relaţional se poate transforma într-una echivalentă din algebra relaţională şi invers. Codd a introdus şase operatori de bază (reuniunea, diferenţa, produsul cartezian, selecţia, proiecţia, joncţiunea) şi doi operatori derivaţi (intersecţia şi diviziunea). Ulterior au fost introduşi şi alţi operatori derivaţi (speciali).

În acest context, **operatorii** din algebra relaţională pot fi grupaţi în două categorii: **pe mulţimi** şi **speciali**.

Operatori pe mulţimi (R1, R2, R3 sunt relaţii (tabele)) sunt:

- **Reuniunea**. *R3 = R1 U R2*, unde R3 va conţine tupluri din R1 sau R2 luate o singură dată;
- **Diferenţa**. *R3 = R1 \ R2*, unde R3 va conţine tupluri din R1 care nu se regăsesc în R2;
- **Produsul cartezian**. *R3 = R1 × R2*, unde R3 va conţine tupluri construite din perechi (x1,x2), cu x1 aparţine R1 şi x2 aparţine R2;
- **Intersecţia**. *R3 = R1 ∩ R2*, unde R3 va conţine tupluri care se găsesc în R1 şi R2 în acelaşi timp;

Operatori relaţionali speciali sunt:

- **Selecţia**. Din R1 se obţine o subtabelă R2, care va conţine o submulţime din tuplurile iniţiale din R1 ce satisfac o condiţie. 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.
- **Proiecţia**. Din R1 se obţine o subtabelă R2, care va conţine o submulţime din atributele iniţiale din R1 şi fără tupluri duplicate. Numărul de atribute din R2 este mai mic decât numărul de atribute din R1.
- **Joncţiunea** este o derivaţie a produsului cartezian, ce presupune utilizarea unui calificator care să permită compararea valorilor unor atribute din R1 şi R2, iar rezultatul în R3. R1 şi R2 trebuie să aibă unul sau mai multe atribute comune care au valori comune.

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

1.4.11 Normalizarea

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

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

- *Anomalia de ştergere* semnifică faptul că ştergând un tuplu dintr-o tabelă, pe lângă informaţiile care trebuie şterse, se pierd şi informaţiile utile existente în tuplul respectiv;
- *Anomalia de adăugare* semnifică faptul că nu pot fi incluse noi informaţii necesare într-o tabelă, deoarece nu se cunosc şi alte informaţii 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 şi prin imprimarea unor facilităţi sporite privind manipularea datelor.

Normalizarea utilizează ca metodă descompunerea (top-down) unei tabele în două sau mai multe tabele, păstrând informaţii (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 tabelelor la această formă normală.

1.4.11.1 Forma Normală 1 (FN1 sau 1NF)

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

O tabelă în **FN1** prezintă încă o serie de anomalii de actualizare datorită eventualelor dependenţe funcţionale 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 şi numai dacă este în **FN1** şi fiecare atribut noncheie al tabelei este dependent funcţional complet de cheie. Un atribut B al unei tabele depinde funcţional de atributul A al aceleiaşi tabele, 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 funcţional complet de un ansamblu de atribute A în cadrul aceleiaşi tabele, dacă B este dependent funcţional 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 dependenţe tranzitive. Eliminarea dependenţelor incomplete se face prin descompunerea tabelei iniţiale în două tabele, ambele conţinând 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 tabelelor 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 şi numai dacă este în **FN2** şi fiecare atribut noncheie depinde în mod netranzitiv de cheia tabelei. Într-o tabelă T, fie A,B,C trei atribute cu A cheie. Dacă B depinde de A (A → B) şi C depinde de B (B → C) atunci C depinde de A în mod tranzitiv.

Eliminarea dependenţelor tranzitive se face prin descompunerea tabelei iniţiale în două tabele, ambele conţinând atributul intermediar (B). O tabelă în **FN3** prezintă încă o serie de anomalii de actualizare, datorate eventualelor dependenţe multivaloare.

1.4.11.4 Forma Normală Boyce Codd (FNBC sau BCNF)

O definiţie 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 faţă de care alte atribute sunt complet dependente funcţional.

1.4.11.5 Forma Normală 4 (FN4 sau 4NF)

O tabelă este în **FN4** dacă şi numai dacă este în **FN3** şi nu conţine două sau mai multe dependenţe multivaloare. Într-o tabelă T, fie A,B,C trei atribute. În tabela T se menţine dependenţa multivaloare A dacă şi numai dacă mulţimea valorilor lui B ce corespunde unei perechi de date (A,C), depinde numai de o valoare a lui A şi este independentă de valorile lui C.

1.4.11.6 Forma Normală 5 (FN5 sau 5NF)

O tabelă este în **FN5** dacă şi numai dacă este în **FN4** şi fiecare dependenţă joncţiune este generată printr-un candidat cheie al tabelei. În tabela T (A,B,C) se menţine dependenţa joncţiune (AB, AC) dacă şi numai dacă T menţine dependenţa multivaloare A -->> B sau C. Dependenţa multivaloare este caz particular al dependenţei joncţiune. Dependenţa funcţională este caz particular al dependenţei multivaloare.

1.4.12 Concluzii

În concluzie, în această primă lecţie au fost prezentate şi explicate mai multe noţiuni teoretice fundamentale pentru a înţelege de ce folosim baze de date relaţionale şi ce noţiuni sunt întâlnite frecvent în lucrul cu baze de date relaţionale. Pe lângă definirea unor noţiuni de bază, a fost abordat şi subiectul normalizării unei baze de date, precum şi prezentarea modelului relaţional.

Lecţia următoare se va axa, în continuare, pe prezentarea unor noţiuni teoretice legate de tipurile de relaţii întâlnite între tabele, dar va conţine şi exemple concrete de proiectare a bazelor de date, precum şi o prezentare a mediilor de lucru folosite pentru a avea instalat un server de baze de date **MySQL** şi 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 şi a metodelor de prelucrarea datelor, spre deosebire de utilizarea bazei de date, care se referă la informaţiile stocate în baza de date.

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

Proiectarea unei baze de date urmăreşte obţinerea următoarelor calităţi:

- **Corectitudine** – reprezentarea cât mai fidelă în baza de date a modului obişnuit de lucru cu datele în sistemul real;
- **Consistenţă** – informaţiile corespunzătoare obiectelor cu care se lucrează în baza de date (nume, definire, relaţii) să nu conţină contradicţii;
- **Distribuire** – informaţiile să poată fi utilizate de aplicaţii multiple şi să poată fi accesate de mai mulţi utilizatori, aflaţi în diferite locuri, utilizând medii diverse
- **Flexibilitate** – facilităţi de adăugare de componente care să reflecte cereri noi de informaţii, să îmbunătăţească performanţele sau să adapteze datele pentru eventuale modificări.

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

- Structura datelor prin definirea unor domenii şi a relaţiilor (atribute, tupluri, chei primare);
- Integritatea datelor prin impunerea unor restricţii;
- Prelucrarea datelor prin operaţii din algebra relaţională sau calculul relaţional.

Modelul relaţional se bazează pe noţiunea matematică de relaţie aşa cum este ea definită în teoria mulţimilor, şi anume ca o submulţime a produsului cartezian a unei liste finite de mulţimi numite domenii.

Algebra relaţională constă dintr-o colecţie de operatori ce au ca operanzi relaţii. Rezultatul aplicării unui operator la una sau două relaţii este tot o relaţie.

Noţiunile de model relaţional şi algebră relaţională au fost discutate în primul capitol al acestui curs.

Proiectarea structurii bazei de date relaţionale (**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 ţinând cont de două aspecte: cerinţele aplicaţiei şi performanţele tehnice ale **SGBD**-ului.

Cerinţele aplicaţiei se referă la: volumul de date estimat a fi memorat şi prelucrat în **BDR**; complexitatea problemei de rezolvat; ponderea şi frecvenţa operaţiilor de intrare/ieşire; condiţiile privind protecţia datelor; operaţiile necesare (încărcare/validare, actualizare, regăsire etc.); particularităţile activităţii pentru care se realizează baza de date.

Performanţele tehnice ale **SGBD**-ului se referă la: modelul de date pe care-l implementează; ponderea utilizării **SGBD**-ului pe piaţă şi tendinţa; configuraţia de calcul minimă cerută; limbajele de programare din **SGBD**; facilităţile de utilizare oferite pentru diferite categorii de utilizatori; limitele **SGBD**-ului; optimizările realizate de **SGBD**; facilităţile tehnice; lucrul cu mediul distribuit şi concurenţa de date; elementele multimedia; posibilitatea de autodocumentare; instrumentele specifice oferite.

Proiectarea **bazelor de date relaţionale (BDR)** se realizează prin proiectarea schemelor **BDR** şi proiectarea modulelor funcţionale specializate.

Schemele bazei de date sunt: *conceptuală*, *externă* şi *internă*.
a) *Proiectarea schemei conceptuale* porneşte de la identificarea setului de date necesar sistemului. Aceste date sunt apoi integrate şi structurate într-o schemă a bazei de date. Pentru acest lucru se parcurg următorii paşi:

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

b) *Proiectare 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 obţinute se structurează logic în subscheme ţinând cont de facilităţile de utilizare şi de cerinţele utilizator. Schema externă devine operaţională prin definirea unor vederi (view-uri) în **SGBD**-ul ales şi acordarea drepturilor de acces. Datele dintr-o vedere pot proveni din una sau mai multe colecţii şi nu ocupă spaţiul fizic.

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

Proiectarea modulelor funcţionale ţine cont de concepţia generală a **BDR**, precum şi de schemele proiectate anterior. În acest sens, se proiectează fluxul informaţional, modulele de încărcare şi de manipulare a datelor, interfeţele specializate, integrarea elementelor proiectate cu organizarea şi funcţionarea **BDR**.

2.2 Realizarea componentelor logice

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

Componentele logice ţin cont de ieşiri, intrări, prelucrări şi colecţiile de date. În paralel cu dezvoltarea programelor de aplicaţii se întocmesc şi documentaţiile diferite (tehnică, de exploatare, de prezentare).

2.3 Punerea în funcțiune şi exploatarea

Se testează funcţiile **BDR** mai întâi cu date de test, apoi cu date reale. Se încarcă datele în **BDR** şi se efectuează procedurile de manipulare, de către beneficiar cu asistenţa proiectantului. Se definitivează documentaţiile aplicaţiei. Se intră în exploatare curentă de către beneficiar conform documentaţiei.

2.4 Întreținerea şi dezvoltarea sistemului

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

2.5 Relații. Tipuri de relații între tabele

În bazele de date relaţionale una dintre cele mai importante noţiuni este cea de relaţie. Tabelele unei baze de date sunt relaţionate între ele. Într-o bază de date relaţională 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 relaţii 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-mulţi* sau *one-to-many* (*1:n*) – unei înregistrări din prima tabelă îi corespund mai multe înregistrări în cealaltă tabelă;
- *mai-mulţi-la-mai-mulţi* sau *many-to-many* (*m:n*) – unei înregistrări din prima tabelă îi corespund una sau mai multe înregistrări din cealaltă tabelă şi reciproc.

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

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

Relaţia *many-to-many* este o relaţie în care avem nevoie de o tabelă intermediară de legătură între cele două tabele (practic relaţia *many-to-many* se descompune în două relaţii *one-to-many*).

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

Această relaţie se descompune în două relaţii de tip *one-to-many,* prin introducerea unei tabele suplimentare care păstrează informaţii de identificare pentru studenţi şi pentru cursurile pe care ei le frecventează.

Înţelegerea modului de relaţionare al tabelelor dintr-o bază de date reprezintă un pas fundamental pentru a putea construi o bază de date optimă atunci când proiectăm o aplicaţie.

2.5.1 Exemplu

Prezentăm în continuare diagrama **entitate-relație** (**ERD)** corespunzătoare unei baze de date în care avem stocate informaţii despre candidaţii la un examen. Sunt reprezentate în această diagramă entităţile, atributele (proprietățile, caracteristicile) fiecărei entităţi, precum şi relaţiile existente între aceste entităţi.
Urmează, ulterior, o prezentare detaliată a fiecărei entități care este modelată în această diagramă.

Figura 2.1. Diagrama ERD iniţială

Tabelele de mai jos prezintă detaliat entităţile modelate în diagramă:

Entitatea: **CALITATE**				
Atribut	*Tipul de date folosit*	*Atribut obligatoriu*	*Identificator unic*	*Semnificaţia*
id_calitate	Numeric	Da	Da	ID-ul ce este asociat calităţii
denumire	Şir de caractere	Da	-	Denumirea calităţii deţinută de profesor în comisie: evaluator, secretar, preşedinte, etc
atributii	Şir de caractere	-	-	Atribuţiile ce decurg din calitatea deţinută: ex: evaluarea elevilor, organizarea examenului etc.

Entitatea: **PROFESOR**				
Atribut	*Tipul de date folosit*	*Atribut obligatoriu*	*Identificator unic*	*Semnificaţia*

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

Entitatea: **SUBCOMISIE**				
Atribut	*Tipul de date folosit*	*Atribut obligatoriu*	*Identificator unic*	*Semnificaţia*
id_subcomisie	Numeric	Da	Da	Id-ul asociat comisiei (ex. 100)
denumire	Şir de caractere	Da	-	Denumirea comisiei (ex. Comisia 1, Comisia 2 etc.)

Entitatea: **CLASA**				
Atribut	*Tipul de date folosit*	*Atribut obligatoriu*	*Identificator unic*	*Semnificaţia*
id_clasa	Numeric	Da	Da	Id-ul asociat clasei (ex. 100)
denumire	Şir de caractere	Da	-	Denumirea clasei (ex. XII A, XII B etc.)

2.5.2 Relaţii între entităţi

Diagrama **ERD** evidenţiază şi relaţiile existente între entităţile modelate. O relaţie între două entităţi arată că există o dependenţă, o legătură naturală între conceptele reprezentate de aceste entităţi. Este evidentă relaţia dintre entităţile CANDIDAT şi CLASĂ : un candidat este un elev ce este înregistrat într-o anumită clasă, iar o clasă este formată din mai mulţi elevi.

Relaţia dintre entitatea A şi entitatea B se defineşte prin:

- denumirea relaţiei: un verb ce sugerează dependenţa dintre cele două entităţi
- opţionalitatea relaţiei: este necesar să stabilim dacă *trebuie* sau *poate* să existe corespondenţă între cele două entităţi
- cardinalitatea relaţiei: precizează numărul de instanţe ale entităţii B ce sunt puse în corespondenţă cu o instanţă a entităţii A.

Relaţia dintre două entităţi este bidirecţională, dar nu simetrică: dacă există o relaţie între A şi B, există şi o relaţie între B şi A, dar nu aceeaşi.

2.6 Convenţii de reprezentare a relaţiilor

1. Linia ce uneşte entităţile relaţionate e formată din două segmente distincte. Tipul liniei ce pleacă de la entitatea A către entitatea B relevă opţionalitatea relaţiei A→B: dacă linia este continuă, relaţia este obligatorie – „trebuie”, iar dacă este discontinuă, relaţia este opţională – „poate”.
2. Denumirea relaţiei A→B este poziţionată lângă entitatea A, deasupra sau dedesubtul liniei de opţionalitate.
3. Cardinalitatea relaţiei A→B se reprezintă astfel: linia de A la B se termină cu o linie simplă, în cazul în care o instanţă a entităţii A este pusă în corespondenţă cu o singură instanţă a entităţii B, şi are forma unui „picior de cioară” (o ramificație în 3 a liniei de relaționare) în cazul în care o instanţă a entităţii A este pusă în corespondenţă cu mai multe instanţe ale entităţii B.

Exemplu:

Figura 2.2 Relaţia dintre entităţile CLASA şi CANDIDAT

Relaţia CLASA→ CANDIDAT

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

Relaţia CLASA→ CANDIDAT se citeşte: „O CLASA trebuie *să aibă* unul sau mai mulţi **CANDIDATI**”.

Relaţia CANDIDAT→ CLASA

- denumirea: **aparţine**
- opţionalitatea: **poate** (e posibil să existe candidaţi din seriile anterioare care să nu mai aparţină vreunei clase)
- cardinalitatea: **una şi numai una**

Relaţia CANDIDAT →CLASA se citeşte: „Un CANDIDAT poate *aparţine* unei singure **CLASE**”.

O clasificare a relaţiilor dintre entităţile A şi B foloseşte cardinalitatea relaţiilor A →B şi B →A.

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

Relaţia dintre entităţile CLASA şi CANDIDAT este de tipul „one-to-many”.

Entităţile relaţionate	*Citirea relaţiei*	*Tipul relaţiei*
PROFESOR, CALITATE	Un PROFESOR trebuie să aibă o singură CALITATE.	One to many

	O CALITATE poate fi deţinută de unul sau mai mulţi PROFESORI.	
SUBCOMISIE, PROFESOR	O SUBCOMISIE trebuie să fie formată din unul sau mai mulţi 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 mulţi CANDIDATI. Un CANDIDAT poate aparţineunei 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 mulţi CANDIDAŢI.	One to many
CANDIDAT, PROBA	Un CANDIDAT poate susţine una sau mai multe PROBE. O PROBĂ poate fi susţinută de unul sau mai mulţi CANDIDAŢI.	Many to many

Se observă relaţia de tip *many-to-many* dintre entităţile CANDIDAT şi PROBA. Relaţiile de acest tip nu pot fi implementate în practică în niciun sistem de gestiune a bazelor de date.

O relaţie *many to many* dintre entitatea A şi entitatea B este descompusă prin adăugarea unei noi entităţi C denumită *entitate de intersecţie* şi construirea de relaţii *one to many* între noua entitate C şi entităţile iniţiale A şi B.

Entităţii de intersecţie i se poate atribui o denumire naturală, ce are o semnificaţie concretă în legătură cu entităţile iniţiale, sau, în lipsa acesteia, o denumire artificială care să combine denumirile entităţilor iniţiale.

În cazul nostru, entitatea de intersecţie este denumită NOTA, deoarece legătura între un candidat şi o probă a examenului este nota obţinută de acesta la acea probă. O denumire artificială ce putea fi folosită este PROBA_CANDIDAT, însă putem recunoaşte că nu este cea mai fericită alegere.

Figura 2.3. Descompunerea relaţiilor „many to many”

Se observă că se păstrează vechile opţionalităţi în partea dinspre entităţile originale, iar relaţiile care pleacă din entitatea de intersecţie sunt întotdeauna obligatorii în această parte.

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

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

Există două posibilităţi:

1) Să se creeze un nou **identificator unic artificial**, de ex: *id_nota*, atribut numeric Valorile acestui identificator unic nu au o semnificaţie concretă pentru o instanţă a acestei entităţi (o notă), singura preocupare fiind asigurarea unicităţii (evitarea duplicatelor). Această metodă este preferată deseori pentru simplitatea implementării, în condiţiile utilizării secvenţelor de numere generate automat (*sequences*).
2) Să se construiască un **identificator unic compus** care să includă identificatorii unici ai entităţilor relaţionate, la care eventual să se adauge atribute proprii entităţii de intersecţie. În acest caz relaţiile cu entităţile de la care s-au utilizat identificatori unici în construcţia UID-ului entităţii de intersecţie trebuie barate către entitatea de intersecţie, iar atributul propriu ce a fost inclus în **UID**-ul compus trebuie precedat de caracterul „#”.

Se observă că **UID**-ul entităţii NOTA este unul compus, format din **UID**-ul *id_candidat* preluat de la entitatea CANDIDAT, **UID**-ul *id_proba* preluat de la entitatea PROBA şi 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 şi se doreşte înregistrarea acestei situaţii, altfel atributele împrumutate în **UID** de la entităţile relaţionate ar fi suficiente pentru unicitate.

Figura 2.4. Definirea UID-ului unic compus al entităţii NOTA

Diagrama **ERD** în forma actuală permite înregistrarea unei note sau a mai multor note obţinute de un candidat la o anumită probă, însă implică discuţii: dacă se înregistrează o singură notă obţinută 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 şti ce notă a acordat fiecare profesor unui candidat la proba respectivă.

Rezultă necesitatea de a relaţiona entitatea NOTA cu entitatea PROFESOR, pentru a întregii informaţiile privitoare la notă.

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

Adăugarea acestei relaţii creează în diagramă o buclă ce creează posibilitatea apariţiei unor relaţii ciclice, *redundante*, situaţie ce trebuie evitată.

O relaţie între două entităţi A şi B este considerată redundantă dacă relaţia se poate deduce din două relaţii A → C şi C → B create anterior.

Un exemplu de relaţie redundantă este prezentat în figura următoare:

Figura 2.5. Relaţie redundantă

Dacă un oraş e împărţit în unul sau mai multe cartiere, şi un cartier e la rândului împărţit în mai multe zone rezidenţiale, se poate deduce că oraşul este împărţit în zone rezidenţiale, fără a fi necesar să marcăm acest lucru în diagramă. Ar apărea o relaţie ciclică, redundantă.

Relaţia PROFESOR → NOTA ar putea fi considerată redundantă, deoarece din diagramă rezultă faptul că un candidat aparţine 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 relaţie directă între CANDIDAT şi NOTA. Pentru a destrăma bucla ar trebui eliminată o relaţie.

Dacă am elimina relaţia PROFESOR → NOTA, neexistând o ierarhie între relaţiile PROFESOR→SUBCOMISIE→CLASA→CANDIDAT→NOTA, nu am ştii ce profesor a acordat nota.

Dacă eliminăm relaţia PROFESOR→SUBCOMISIE, nu am cunoaşte (sau ar fi dificil de aflat) fiecare profesor cărei subcomisii aparţine.

Soluţia este păstrarea diagramei în forma prezentă, nefiind de fapt o situaţie de redundanţă, deoarece numele relaţiilor nu sugerează că dacă un profesor este alocat unei subcomisii, el evaluează neapărat.

2.7 Limbajul SQL. Introducere

Primele sisteme de baze de date relaţionale au apărut în 1970. Cele mai populare **SGBD**-uri relaţionale sunt: **Oracle**, **Microsoft SQL Server**, **MySQL**. Toate aceste sisteme de baze de date relaţionale 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 - **S**tructured **Q**uery **L**anguage este un limbaj de baze de date realizat pentru a extrage informaţii şi a administra bazele de date relaţionale. Limbajul **SQL** a devenit standard **ANSI** (**A**merican **N**ational **S**tandards **I**nstitute) în 1986. Fiecare sistem de management al bazei de date (**RDBMS** - **R**elational **D**atabase **M**anagement **S**ystem) are propria versiune de limbaj **SQL**, bazată pe standardul **SQL**. Astfel, limbajul **SQL** folosit în **MySQL**, fată de limbajul **SQL** folosit în PostgreSQL sau Oracle, deşi asemănătoare, au elemente distincte, specifice acelui **RDBMS**.

MySQL este o aplicaţie comercială pentru managementul bazelor de date relaţionale (pe scurt un **RDBMS**) foarte populară, mai ales în dezvoltarea aplicaţiilor 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** şi 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 preferinţă a programatorilor dezvoltatori în **PHP** pentru baze de date este **MySQL**.

În plus, **PHP** are extensii (set de funcţii) pentru a lucra şi 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 şi manipulează datele, şi răspunde clienţilor ce se conectează la acesta pentru a cere informaţii sau pentru a trimite cereri de altă natură (adăugări, modificări, etc). Serverul **MySQL** şi clientul **MySQL** folosit pentru interogare

pot fi instalate pe acelaşi calculator, dar nu neapărat. Dacă lucrăm local (pe calculatorul propriu) şi folosim un program ca **WAMP** server, atât serverul **MySQL** cât şi 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.

Aşadar, **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 noţiunile pe care le vom învăţa. 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 instalze 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** şi unul de baze de date **MySQL**. Acronimul WAMP vine de la **W**indows **A**pache **M**ySQL **P**HP.
Adresa de la care se poate descărca acest program este următoarea: http://www.wampserver.com/en/

Corespondentul **WAMP** pentru sistemul de operare Linux, este un program numit LAMP (**L**inux **A**pache **M**ySQL **P**HP).

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

Toate aceste programe pot fi folosit şi atunci când realizăm aplicaţii web pe calculatorul propriu în limbajul **PHP** şi 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 execuţie acest program.

La pornire, pictograma acestei aplicaţii se plasează în partea dreaptă a barei de start şi, atunci când toate serviciile oferite sunt pornite, are culoarea verde.
În continuare iată şi fereastra **WAMP** care se deschide la clic pe această pictogramă:
În această imagine se observă printre serviciile oferite de **WAMP** şi **MySQL**.

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

2.9.2 Clientul de baze de date HeidiSQL

De asemenea, vom instala şi 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 execuţie se va deschide următoarea fereastră:

La Network Type opţiunea 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, conectatea fără parolă nu este recomandată pe un server real din motive de securitate).
Se apasă butonul Open şi se deschide fereastra următoare:

În această fereastră, în partea stângă se observă bazele de date disponbile, iar în partea dreaptă vedem tabelele bazei de date selectate, dacă există sau avem tab-ul *Query* care permite scrierea de instrucţiuni ***MySQL*** şi rularea acestora prin acţionarea 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 vizulizarea 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 lecţie a dezvoltat conceptul de proiectare a unei baze de date relaţionale. Am prezentat în cadrul ei exemple concrete de realizare a design-ului unei baze de date. Conceptul a fost prezentat şi explicat pe larg, cu exemple concrete. Tot în cadrul acestei lecţii s-a realizat şi o introducere în limbajul **SQL**.

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

În următoarea lecţie se va trece la prezentarea sintaxei **SQL**. Vom discuta pe larg despre **L**imbajul de **D**escriere a **D**atelor (**LDD**) şi despre comenzile (instrucţiunile) acestui limbaj care se referă la structura bazei de date şi a tabelelor componente. Va fi prezentată sintaxa precum şi 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 noţiunile prezentate în capitolele anterioare şi raportându-ne concret la o bază de date MySQL, avem următoarele corespondenţe:

- 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)** conţine comenzi pentru:

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

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

Aşadar, pe parcursul acestui curs vom întâlni în foarte multe rânduri termenii de ***instrucţiune SQL***, ***comandă SQL*** sau ***frază SQL***. De asemenea, trebuie precizat că, spunem la modul generic ***comandă SQL*** sau ***frază SQL*** sau ***instrucţiune SQL***, deşi, în cazul nostru, este limpede că ne vom referi la ***instrucţiuni/comenzi/fraze MySQL***. După cum am precizat şi în lecţia precedentă, **MySQL** este un **S**istem de **G**estiune al **B**azelor de **D**ate (**SGBD**) bazat pe standardul **SQL (S**tructured **Q**uery **L**anguage**).**

Deşi prezintă particularităţi, ca de altfel orice **SGBD**, totuşi instrucţiunile **MySQL** sunt foarte asemănătoare cu cele ale standardului **SQL**. Din acest motiv, în mod uzual, vorbim de instrucţiuni **SQL**. Cu alte cuvinte, putem spune că **SQL** este un nucleu de la care s-au dezvoltat aceste aplicaţii complexe utilizate pentru interacţiunea cu bazele de date, denumite **SGBD**-uri.

Aşa cum spuneam, în **MySQL**, ca de altfel în orice alt **SGBD**, pentru fiecare instrucţiune/comandă avem o sintaxă, adică un set de reguli de scriere care trebuie respectat pentru a nu fi generate erori la rularea instrucţiunilor.
Astfel, există un set de **cuvinte rezervate** ale limbajului care definesc anumite acţiuni şi care trebuie specificate în cadrul unei instrucţiuni **MySQL**. Aceste cuvinte speciale (rezervate) se numesc **cuvinte cheie**. Atunci când scriem o instrucţiune **MySQL** trebuie, aşadar, să folosim aceste cuvinte cheie, plus alte cuvinte (nume de tabele, nume de câmpuri, valori, operatori, etc.) scrise într-o anumită ordine.

Important de reţinut este şi faptul că orice instrucţiune **MySQL** se încheie cu **caracterul „;”**.

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

Acest fapt uşurează în mare parte scrierea unei instrucţiuni. O altă menţiune 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 execuţia unor comenzi ce conţin aceste denumiri. Un câmp denumit astfel ar trebui prefixat de numele tabelei în fiecare instrucţiune pentru a nu fi eroare, deci s-ar complica mult codul instrucţiunii scrise.

3.2 Crearea unei baze de date

Comanda care se foloseşte pentru crearea unei baze de date în MySQL este:
CREATE DATABASE *nume_bd*;
unde *nume_bd* reprzintă numele pe care vrem să îl aibă baza de date.
Denumirea bazei de date poate să conţină doar caractere alfanumerice şi semnul „_”.

3.3 Ştergerea unei baze de date

Comanda care se foloseşte 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 instrucţiuni se foloseşte baza de date specificată (se pot realiza operaţii pe această bază de date). Comanda este utilă atunci când avem mai multe baze de date.
Fiecare instrucţiune MySQL se încheie cu caracterul „**;**”.

3.5 Crearea unei tabele

Comanda care se foloseşte 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) tabelei, *tip_dată* reprezintă tipul de dată al câmpului respectiv (de exemplu: numeric, șir de caractere, dată calendaristică, etc.), *modificatori* este opţional la fel ca şi *restricţiile* ce se pot aplica pe anumite câmpuri. Modificatorii reprezintă anumite opțiuni sau restricții ce se pot aplica asupra coloanelor din tabelă.

3.6 Ştergerea unei tabele

Comanda care se foloseşte pentru a şterge o tabelă dintr-o bază de date este următoarea:
DROP TABLE *nume_tabelă*;

3.7 Modificarea unei tabele

Comanda care se foloseşte 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 definiţiei unui câmp:

ALTER TABLE *nume_tabelă* **CHANGE** *nume_câmp nume_câmp* definiţie_câmp;

sau

ALTER TABLE *nume_tabelă* **MODIFY** *nume_câmp* definiţie_câmp;

- pentru adăugarea unui câmp într-o tabelă

ALTER TABLE *nume_tabelă* **ADD** *nume_câmp* definiţie_câmp;

- pentru ştergerea 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
- şiruri 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 spaţiu mai puţin pe disc.

Fiecărui tip de dată i se specifică şi lungimea, de exemplu dacă avem valori de la 1 la 100 într-un câmp putem aloca 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 conţine preţul unui produs poate fi definit de tip **DOUBLE**(5,2).
În paranteză este trecut numărul total de cifre al preţului (5) respectiv numărul obligatoriu de zecimale care va fi afişat (2). Prin urmare, datele introduse în câmpul preş 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 şiruri 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 şir de caractere sau string. Într-o tabelă cu informaţii despre angajaţii unei companii, de exemplu, numele angajaţilor poate fi ţinut într-un câmp nume de tip **VARCHAR**(70), între paranteze fiind trecută dimensiunea maximă pe care o poate avea valoarea introdusă în acest câmp.

Aşadar, 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, şi un câmp prenume de tip **VARCHAR**, de dimensiune 100 de caractere, adică **VARCHAR**(100).

Tipurile de date **CHAR** şi **VARCHAR** acceptă şi 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 diferenţă între aceste două tipuri este că şirul dintr-un tip **CHAR** va fi stocat întotdeauna ca un şir cu lungimea maximă a coloanei, folosind spaţii pentru completare, dacă şirul introdus este mai mic decât lungimea coloanei.

Tipurile de date folosite pentru stocare text sunt următoarele:
TINYTEXT – un şir cu lungime maximă de 255 de caractere;
TEXT – un şir cu o lungime amximă de 65 535 de caractere;
MEDIUMTEXT – un şir cu o lungime maximă de 16 777 215 de caractere;
LONGTEXT – un şir 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 foloseşte 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 conţine conţinutul (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 şi pentru câmpurile cu texte lungi.

Tipurile de date binare întâlnite în cadrul MySQL sunt următoarele:
TINYBLOB – stochează până la 255 bytes;
BLOB (**B**inary **L**arge **Ob**ject) **–** stochează până la 64 KB;
MEDIUMBLOB – stochează până la 16 MB;
LONGBLOB – 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 şiruri binare.

Ele nu au un set de caractere iar sortarea şi compararea lor se bazează pe valorile numerice ale octeţilor 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-secunda*;
DATETIME – stochează data şi ora în formatul *an-lună-zi ora-minut-secundă*;
TIMESTAMP – este util la înregistrarea unor operaţii precum inserare sau actualizare pentru că reţine implicit data efectuării ultimei operaţii.

Singurul format în care **MySQL** păstrează şi afişează 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' pînă la '9999-12-31'. Dacă avem într-o tabelă a unei baze de date stocată data naşterii 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 şi 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 şi 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 şi a unei sintaxe specifice.

Modificatorii ce pot fi întâlniţi în definiţiile de descriere ale unui câmp sunt:
NOT NULL – modificator sau constrângere care stabileşte 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 defineşte acel câmp ca fiind cheie primară a unei tabele;
FOREIGN KEY – constrângere care defineşte o cheie externă pentru o tabelă;

UNIQUE – constrângere de unicitate care impune valori unice pentru cîmpurile care au definit acest modificator;
INDEX – constrângere care aplică un index pe un câmp al unei tabele.

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

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

Deci instrucţiunea de creare este următoarea:

```
CREATE DATABASE bibilioteca;
```

Prezentăm în continuare comanda de creare a unei tabele, autori ce conţine informaţii despre autorii cărţilor din această bibliotecă.

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

Aşadar, avem o instrucţiune de creare a unei tabele din baza de date, este vorba de o tabelă simplă cu doar 2 câmpuri ce conţine numele autorilor cărţilor. Primul câmp, *id_autor*, conţine valori care identifică în mod unic o înregistrare, deci acest câmp este utilizat drept cheie primară a tabelei.

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

Al doilea câmp, *nume*, conţine un şir de caractere de lungime maximă 200 de caractere ce va stoca numele şi prenumele fiecărui autor.

3.11 Chei externe

Instrucţiunea prin care se aplică restricţia de cheie externă este următoarea:
FOREIGN KEY (*nume_câmp*$_1$) **REFERENCES** *nume_tabelă*(*nume_câmp*$_2$)

FOREIGN KEY şi **REFERENCES** sunt cuvinte cheie în timp ce ***nume_câmp*$_1$**reprezintă câmpul din tabelă care este cheie externă în tabelă, în timp ***nume_câmp*$_2$** reprezintă câmpul la care face referire, adică cheia primară din tabela *nume_tabelă*.

3.12 Index

Introducem în continuare noţiunea de index. Indecşii sunt folosiţi pentru sortarea logică a datelor în vederea îmbunătăţirii vitezei operaţiilor de căutare şi sortare.

Indecşii dintr-o bază de date funcţionează în maniera următoare: datele din cheile primare sunt întotdeauna sortate; este o operaţie pe care programul **SGBD** o execută. Deci, regăsirea anumitor rânduri în funcţie de cheia primară este întotdeauna o operaţie rapidă şi eficientă.

Un index se poate definii pe una sau mai multe coloane. Indecşii îmbunătăţesc performanţele operaţiilor de regăsire dar le degradează pe acelea ale operaţiilor de inserare, modificare şi ştergere a datelor. Când sunt executate aceste operaţii, programul **SGBD** trebuie să actualizeze indexul în mod dinamic. Datele din index pot ocupa o cantitate mare de spaţiu de stocare.

Atunci când se defineşte un index se creează un fişier de index, iar în momentul în care este executată o instrucţiune de interogare pe câmpul indexat, se face practic o căutare în fişierul de index, din acest motiv avem o viteză foarte mare de execuţie. 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 tabelei autori în autori_noi:

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

Aceeaşi comandă poate fi scrisă şi 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 şi o comandă care adaugă un nou câmp în această tabelă, câmpul *prenume*:

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

De asemenea, prezentă şi instrucţiunea pentru ştergerea unui cîmp:

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

Prezentăm în continuare alte exemple de utilizare a instrucţiunilor din limbajul de desciere a datelor. Astfel, vom realiza o instrucţiune de creare a unei baze de date, apoi vom crea 2 tabele în această bază de date şi vom aplica diverse constrângeri.

Presupunem crearea unei baze de date în care se păstrează informaţii despre angajaţii şi departamentele unei companii. Aşadar, vom avea o bază de date pe care o vom denumi companie şi în această bază de date vom crea două tabele pentru evidenţa departamentelor, respectiv a angajaţilor.

Instrucţiunea pentru crearea bazei de date *companie*:

```
CREATE DATABASE companie;
```

Instrucţiunea pentru crearea tabelei *departamente*:

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

Instrucţiunea pentru crearea tabelei *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 definiţia de creare a tabelei *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.

Aşa cum am precizat anterior, o cheie externă poate fi definită în instrucţiunea de creare a unei tabele, după ce au fost enumerate toate coloanele tabelei sau poate fi creată prin instrucţiuni de modificare a structurii unei tabele din baza de date.

Pentru a prezenta şi cea de-a doua modalitate, mai întâi vom elimina (şterge) constrângerea de cheie externă aplicată câmpului *id_departament* din tabela *angajati*.

Instrucţiunea pentru eliminarea unei constrângeri de tip **FOREIGN KEY** este următoarea:

```
ALTER TABLE angajati DROP FOREIGN KEY nume_cheie;
```

Fiecare **FOREIGN KEY** primeşte automat un nume dacă noi nu am asociat un nume în definirea constrângerii. Pentru a defini un nume unei chei externe, în faţa instrucţiunii de creare şi 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 instrucţiune:

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

Aşadar, instrucţiunea de ştergere 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 şi 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 tabelei *angajati* ar fi aparut o problemă de integritate a datelor în momentul în care încercam 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 instrucţiuni de modificare a structurii unei tabele din baza de date. De exemplu, eliminarea cheii primare din tabela *departamente* se face cu instrucţiunea:

```
ALTER TABLE departamente DROP PRIMARY KEY;
```

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

```
ALTER TABLE departamente ADD PRIMARY KEY (id_departament);
```

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

```
ALTER TABLE angajati CHANGE prenume prenume VARCHAR(150);
```

Ştergerea unui câmp dintr-o tabelă se poate face cu următoarea instrucţiune:

```
ALTER TABLE departamente DROP manager;
```

Adăugarea unui câmp într-o tabelă se poate face cu următoarea instrucţiune:
ALTER TABLE departamente **ADD** manager VARCHAR(150) **NOT NULL**;
De asemenea, observăm în instrucţiunea anterioară şi 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 instrucţiunea:
ALTER TABLE angajati **CHANGE** cnp cnp VARCHAR(13) **NOT NULL**;

3.14 Concluzii

Aşadar, în această lecţie, au fost prezentate instrucţiuni şi 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 lecţia următoare, conţine instrucţiuni 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

Creaţi o **bază de date** în care să se păstreze evidenţa clienţilor unei companii, precum şi 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** reţine informaţii de bază despre clienţii companiei: **id_client**, **nume**, **prenume**, **număr de telefon**,**localitate**. Tabela **facturi** conţine informaţii despre facturile clienţilor: **id_factura**, **data_factura**, **valoare**, **clientul**caruia i-a fost emisă factura.

Cerinţe:

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

8. Scrieţi o instrucţiune **SQL** prin care eliminați constrângerea **FOREIGN KEY** din tabela **facturi**.
9. Scrieţi o instrucţiune **SQL** prin care adăugați o constrângere **FOREIGN KEY** pe tabela **facturi** cu numele **fk_fact**.
10. Scrieţi o instrucţiune **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 conţine comenzile de actualizare a datelor în interiorul tabelelor dintr-o bază de date, precum şi 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** şi **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** şi are următoarea sintaxă:
INSERT INTO *nume_tabelă* (*câmp*$_1$, *câmp*$_2$, ..., *câmp*$_n$) **VALUES** (*valoare*$_1$,*valoare*$_2$, ..., *valoare*$_n$);

Instrucţiunea **INSERT** mai are şi alte forme în care poate fi utilizată. Una dintre ele este aceea în care se vor insera valori în toate coloanele unei tabele și are sintaxa:
INSERT INTO *nume_tabelă* **VALUES** (*valoare*$_1$, *valoare*$_2$, ..., *valoare*$_n$);

Următoarea formă a instrucţiunii 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* (*câmp*$_1$, *câmp*$_3$) **VALUES** (*valoare*$_1$, *valoare*$_3$);

Urmează o formă diferită de celelalte a comenzii **INSERT**, asemănătoare, vom vedea cu sintaxa comenzii de actualizare **UPDATE**:
INSERT INTO *nume_tabelă* **SET** *nume_câmp*$_1$ = *valoare*$_1$, ..., *nume_câmp*$_n$ =*valoare*$_n$;

Ultima formă permite inserearea mai multor înregistrări printr-o singură instrucţiune **INSERT:**
INSERT INTO *nume_tabelă* (*câmp*$_1$, *câmp*$_2$, ..., *câmp*$_n$) **VALUES** (*valoare*$_{11}$,*valoare*$_{12}$, ..., *valoare*$_{1n}$), (*valoare*$_{21}$, *valoare*$_{22}$, ..., *valoare*$_{2n}$), ..., (*valoare*$_{n1}$,*valoare*$_{n2}$, ..., *valoare*$_{nn}$) ;

Comanda pentru actualizarea sau modificarea unei înregistrări este **UPDATE**.
Sintaxa comenzii de actualizare a datelor este următoarea:
UPDATE *nume_tabelă* **SET** *nume_câmp*$_1$ = *valoare*$_1$ *[, ..., nume_câmp*$_n$ = *valoare*$_n$*]*
[**WHERE** condiții]
[**ORDER BY** coloane]
[**LIMIT** număr_rânduri];
Dacă lipseşte clauza **WHERE** înseamnă că se vor actualiza toate înregistrările din tabelă.

Comanda pentru ştergerea datelor dintr-o tabelă este **DELETE**.
Sintaxa comenzii de ştergere a datelor este următoarea:
DELETE FROM *nume_tabelă*
[**WHERE** condiții]

[**ORDER BY** coloane]
[**LIMIT** număr_rânduri];

Dacă lipseşte clauza **WHERE** se vor şterge 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 condiționărilor impuse.
Sintaxa comenzii **SELECT** este următoarea:

SELECT [**DISTINCT**] câmp$_1$, câmp$_2$,...,câmp$_n$ [**FROM** *nume_tabelă*]
[**WHERE** condiţii]
[**GROUP BY** câmp$_1$ [,câmp$_2$ …]]
[**HAVING** condiţii]
[**ORDER BY** câmp$_1$ [**ASC** | **DESC**] [,câmp$_2$ [**ASC** | **DESC**], …]]
[**LIMIT** nr_rânduri];

Aşadar, acestea sunt cele 4 instrucţiuni ce compun limbajul de manipulare a datelor. Ele au o sintaxă relativ simplă şi uşor de înţeles. În continuare vom explica fiecare comandă în parte şi vom prezenta câteva exemple de utilizare practică.

4.2 Instrucțiunea INSERT

Reluăm, în continuare, sintaxa instrucţiunii 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* (*câmp$_1$, câmp$_2$, ..., câmp$_n$*) **VALUES** (*valoare$_1$,valoare$_2$, ..., valoare$_n$*);

Instrucțiunea de adăugare de informaţii 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 tabele și, în această situație, nu mai trebuie menționate coloanele tabelei, 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** (*valoare$_1$, valoare$_2$, ..., valoare$_n$*);
INSERT INTO *nume_tabelă* **SET** *nume_câmp$_1$ = valoare$_1$, ..., nume_câmp$_n$ =valoare$_n$*;

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

INSERT INTO *nume_tabelă* (*câmp$_1$, câmp$_2$, ..., câmp$_n$*) **VALUES** (*valoare$_{11}$,valoare$_{12}$, ..., valoare$_{1n}$*),
(*valoare$_{21}$, valoare$_{22}$, ..., valoare$_{2n}$*), ..., (*valoare$_{n1}$,valoare$_{n2}$, ..., valoare$_{nn}$*) ;

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. Aşa cum am prezentat mai sus, comanda **INSERT** este folosită pentru acastă operaţie.

Considerăm tabela *angajaţi* ce conţine informaţii despre angajaţii unei companii (nume, prenume, data naşterii, data angajării, salariul).
Comanda pentru crearea acestei tabele 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 restricţia **AUTO_INCREMENT** nu este introdusă nici o valoare (apare null), deoarece se va insera automat pentru coloana id un număr întreg care creşte 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 inserearea de valori nule.

În cazul în care una din înregistrări este ştearsă nu se va aloca id-ul ei unei înregistrări nou introdusă în tabelă. În memoria internă se păstează valorea la care a ajuns incrementul.

În câmpul *nume* şi în câmpul *prenume* se introduc şiruri de caractere, în câmpurile *data_nasterii* şi *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 instrucţiunea **INSERT** de introducere a datelor în tabelă, mai ales că nu am aplicat restricţia **NOT NULL** pe câmpurile tabelei (doar cîmpul **id**, care este cheie primară are această restricţie, dar acest câmp se şi auto incrementează şi atunci nu trebuie specificat în comanda **INSERT**).

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

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

Iată şi o instrucţiune care inserează mai multe înregistrări în tabelă (este omisă specificarea coloanelor, deci, se vor insera valori în toate coloanele tabelei, 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 instrucţiune va introduce 3 înregistrări în tabelă. Observăm faptul că lipseşte partea instrucţiunii în care sunt specificate câmpurile în care se introduc date. Acest lucru este posibil deoarece sunt inserate date în toate câmpurile tabelei. Atenţie, 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 execuţia comenzii (de exemplu, dacă încercăm să introducem un şir de caractere într-un câmp de tip int, sau o dată într-un câmp de tip double).

Dacă execuţia instrucţiunii va genera o eroare, datele nu sunt inserate în tabelă, chiar dacă o parte din ele sunt date care corespund tipurilor specificate. Instrucţiunea 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 instrucţiunea **UPDATE**.
Sintaxa unei instrucţiuni **UPDATE** este următoarea:

UPDATE *nume_tabelă* **SET** *nume_câmp*$_1$ = *valoare*$_1$ *[, ..., nume_câmp*$_n$ = *valoare*$_n$*]*
[**WHERE** condiții]
[**ORDER BY** coloane]
[**LIMIT** număr_rânduri];

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

Considerând tabela *angajati* pe care am utilizat-o şi 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 instrucţiune de actualizare (modificare), lipseşte clauza **WHERE** se vor modifica informaţiile din toate înregistrările tabelei. Deci, trebuie să fim atenţi atunci când folosim această instrucţiune de actualizare deoarece, în cele mai multe din cazuri, nu se doreşte actualizarea tuturor înregistrărilor dintr-o tabelă.

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

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

Astfel, considerând că tabela *angajati* conţine câteva sute de înregistrări, dacă dorim actualizarea salariului la valoarea 2500 pentru primii 10 de angajaţi în funcție de vechimea lor vom folosi în instrucţiunea **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 instrucţiunea **UPDATE** nu mai există clauza **WHERE**, dar, totuşi actualizarea nu se face pentru toate înregistrările tabelei, 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 şi această variantă de limitare la un anumit număr de înregistrări, totuşi cea mai folosită formă a instrucţiunii **UPDATE**, este cea care conţine una sau mai multe condiţii care trebuie să fie îndeplinite pentru a realiza actualizarea datelor. În acest mod vom şti cu certitudine că nu au fost actualizate înregistrări care nu îndeplinesc condiţiile dorite pentru a se realiza modificarea.

4.4 Instrucțiunea DELETE

Instrucţiunea folosită pentru ştergerea înregistrărilor din baza de date este **DELETE**.
Sintaxa instrucţiunii de ştergere 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 aceeaşi tabelă pe care am utilizat-o mai înainte, *angajati*, ştergerea înregstrării cu id-ul 3 se face prin următoarea comandă:

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

La fel ca în cazul instrucţiunii **UPDATE** şi, într-o instrucţiunea **DELETE**, dacă lipseşte clauza **WHERE**, care stabileşte condiţia ce trebuie să fie îndeplinită pentru a se executa ştergerea înregistrărilor, se vor şterge toate înregistrările din tabelă. Deci, trebuie folosită cu atenţie această instrucţiune, astfel încât să fim siguri că am stabilit condiţiile necesare a fi îndeplinite pentru a şterge anumite înregistrări.

În general, se evită folosirea instrucţiunii de ştergere din tabelele unei baze de date a unei aplicaţii aflată în utilizare. Comanda **DELETE** va fi utilizată atunci când ştim sigur că datele respective nu ne mai sunt necesare în baza de date.
În concluzie, acestea sunt cele 3 instrucţiuni folosite pentru actualizarea datelor din tabelele unei baze de date. Sunt instrucţiuni cu sintaxă destul de simplă şi cu o logică uşor de înţeles. Dacă instrucţiunea **INSERT** are mai multe forme, în schimb, instrucţiunile **UPDATE** şi **DELETE** au o singură formă asemănătoare şi uşor de înţeles şi 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 şi resetarea auto incrementului

Comanda care se foloseşte pentru a şterge toate datele dintr-o tabelă este următoarea:

```
TRUNCATE TABLE nume_tabelă;
```

Această comandă, pe lângă ștergerea tuturor înregistrărilor din tabelă, va reseta şi 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 Instrucţiunea SELECT

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

Sintaxa acestei comenzi a fost prezentată în prima parte a lecţiei, dar o vom relua şi aici, urmând ca apoi să explicăm fiecare clauză care poate să apară într-o astfel de instrucţiune de interogare a tabelelor dintr-o bază de date.
Aşadar, reluăm sintaxa instrucţiunii de regăsire a datelor, care a fost prezentată și în prima parte a lecției:

SELECT [**DISTINCT**] câmp$_1$, câmp$_2$,...,câmp$_n$ [**FROM** *nume_tabelă*]
[**WHERE** condiţii]
[**GROUP BY** câmp$_1$ [,câmp$_2$ …]]
[**HAVING** condiţii]
[**ORDER BY** câmp$_1$ [**ASC** | **DESC**] [,câmp$_2$ [**ASC** | **DESC**], …]]
[**LIMIT** nr_rânduri];

După cum se observă din prezentarea completă a sintaxei, instrucţiunea **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 sa 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 puţin două informaţii: ce anume dorim să selectăm şi locaţia 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 tabele se foloseşte caracterul asterisc „*****” care reprezintă selectarea tuturor coloanelor dintr-o tabelă.

Instrucţiunea următoare va extrage şi va afişa toate înregistrările din tabela *angajati*:

SELECT * FROM *angajati*;

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

SELECT nume **FROM** *angajati*;

Specificarea mai multor coloane ale unei tabele într-o instrucţiune **SELECT** se face prin separarea câmpurilor (coloanelor) tabelelor prin virgulă. Pentru a selecta numele, prenumele şi salariul angajaţilor stocate în tabela *angajati* se va utiliza următoarea instrucţiune **SELECT**:

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

4.7 Clauza WHERE

În continuare vom descrie clauza **WHERE** a instrucţiunii **SELECT**. Este o clauză opţională, dar este foarte des folosită şi foarte importantă. În cazul în care lipseşte clauza **WHERE** dintr-o interogare, atunci se vor afişa toate înregistrările din coloanele specificate în instrucţiunea **SELECT** din tabela respectivă.

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

4.8 Operatori folosiţi 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 şi 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 au 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 foloseşte este **BETWEEN** valoare_minimă **AND** valoare_maximă;
- **IN** – acest operator testează dacă operandul se regăseşte printre lista de valori care este specificată între paranteze; acest operator este folosit în forma următoare: **IN(valoare$_1$, valoare$_2$,...,valoare$_n$)**; valorile testate cu operatorul **IN** pot fi obţinute şi printr-o instrucţiune **SELECT**, deci poate fi folosit în subinterogări;
- **IS NULL** – verifică dacă valoarea dintr-o coloană a tabelei este **NULL**;
- **IS NOT NULL** – verifică dacă valoarea dintr-o coloană a tabelei nu este **NULL**;

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

Acest operator este folosit pentru a verifica dacă valoarea de tip şir de caractere dintr-o coloană corpespunde cu un şir de caractere specificat sau, putem folosi aici şi caractere de înlocuire. Astfel avem caracterul de înlocuire „**%”** care are semnificaţia că găseşte orice caracter, indiferent de câte ori apare.

De exemplu pentru a găsi toţi angajaţii 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 şi acest caracter înlocuieşte un singur caracter, nici mai mult, nici mai puţin. În schimb caracterul de înlocuire „**%”** poate să substituie un caracter, nici un caracter (zero caractere) sau oricât de multe caractere.

În continuare vom prezenta şi alţi operatori folosiţi în clauze **WHERE** mai complexe în care punem mai multe condiţii, deci combinăm mai multe condiţii simple. Astfel, intervin operatorii logici:

- **AND (&&)** – operatorul „**şi**” logic, va returna **adevărat (1)** dacă toţi operanzii sunt adevăraţi, respectiv **fals (0)** dacă cel puţin unul dintre operanzii este fals;
- **OR (||)** – operatorul „**sau**” logic, va returna **adevărat (1)** dacă cel puţin unul dintre operanzi este adevărat, respectiv **fals (0)** dacă toţi operanzii sunt falşi;
- **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 şi numai unul din aceşti operanzi este adevărat iar celălalt fals, dacă ambii operatori sunt la fel rezultatul returnat va fi **fals (0)**; dacă avem mai mulţi 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 foloseşte 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 conţine 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 funcţie 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 funcţiilor agregate (funcţii de grup), acestea acţionând 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 foloseşte 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 instrucţiune **SELECT** putem avea:

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

În exemplul următor, dacă vrem să obţinem numărul de angajaţi 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 instrucţiune **SELECT**:

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

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

4.10 Clauza HAVING

Dacă într-o instrucţiune **SELECT** folosim funcţii de agregare şi avem nevoie să punem condiţii pe rezultatul obţinut în urma utilizării acestor funcţii, atunci vom folosi clauza **HAVING**. Mai simplu de reţinut, clauza **HAVING** se foloseşte atunci când avem în instrucțiunea de regăsire **SELECT** funcţii 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 funcţii de grup sau funcții de agregare care se întâlnesc în limbajul **SQL** sunt următoarele:

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

Funcţia **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.

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

Funcţia **AVG()** întoarce media aritmetică a valorilor din expresia primită ca argument şi poate primi ca argument o coloană a unei tabele sau o expresie.

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

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

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

Dacă instrucţiunea **SELECT,** în care au fost utilizate funcţii de agregare, nu conţine clauza **GROUP BY**, atunci valoarea funcţiilor de agregare va fi calculată pentru întreaga tabelă specificată în clauza **FROM** a instrucţiunii de interogare a bazei de date.

Pentru exemplul anterior, dacă vrem doar afişarea departamentelor cu cel puţin 2 angajaţi, instrucţiunea **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 instrucţiunea **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 foloseşte cuvântul cheie **AS** urmat de numele asociat acelei expresii, în cazul nostru putem asocia **alias-ul *număr_angajati*** expresiei **COUNT(*id_angajat*).**

Prin urmare, instrucţiunea **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 tabelei rezultat coloanele ***id_dept*** şi ***nr_angajati***. Dacă nu asociem un **alias** expresiei de numărare, coloanele rezultate ar fi ***id_dept*** şi **COUNT(*id_angajat*)**

4.12 Clauza ORDER BY

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

Clauza utilizată pentru ordonarea datelor rezultate în urma unei selecţii este **ORDER BY**, după această clauză se specifică numele câmpului după care se face ordonarea şi tipul de sortare (crescător sau descrescător). Pentru sortare în ordine crescătoare avem cuvâtnul 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**, şi aşa mai departe pentru toate câmpurile din clauză. De asemenea, sortarea se poate face crescător după anumite câmpuri şi 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 toţi angajaţii din baza de date sortaţi după nume şi prenume vom realiza următoarea interogare:

```
SELECT * FROM angajati
```

```
        ORDER BY nume, prenume;
```

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

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

În cazul în care se doreşte 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 şi crescătoare după prenume, adică angajaţii sunt sortaţi după nume în ordine inversă, iar dacă există mai mulţi angajaţi cu acelaşi 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 instrucţiuni **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 instrucţiunea **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 poziţia 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 poziţia de unde începe şi câte înregistrări vor fi returnate de interogare).

Clauza **LIMIT**, atunci când este utilizată, este întotdeauna ultima în cadrul unei instrucţiuni **SELECT**.
Afişarea primilor 10 angajaţi din tabela în care sunt salvaţi, ordonaţi alfabetic după nume, se realizează cu următoarea instrucţiune:

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

Sintaxa acesteia are una dintre următoarele două forme, aşa cum am precizat şi anterior:

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

Important de reţinut este faptul că prima linie este numerotată cu 0. Aşadar, instrucţiunea 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 poziţia 0. Deci, prima înregistrare rezultată în urma unei selecţii se află pe poziţia 0.

Dacă am fi scris următoarea instrucţiune:

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

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

Dacă în tabela noastră presupunem că am avea 100 de înregistrări, afişarea ultimilor 10 angajaţi sortaţi în ordine alfabetică după nume s-ar realiza cu instrucţiunea:

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

Întrucât prima poziţie 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 cerinţa anterioară este **LIMIT *90,10***, iar nu **LIMIT *91,10***. A doua variantă ar fi afişat doar 9 înregistrări, întrucât începând cu linia 91 nu mai există 10 înregistrări ăn tabelă. Deci, atunci când numărul de înregistrări care ar trebui afişate, specificat în clauza **LIMIT** este mai mare decât numrăul de înregistrări care există în tabelă, de la poziţia (linia) dată, atunci se afişează 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 afişare, ci vor fi afişate atâtea câte există.

4.14 Clauza DISTINCT

Într-o tabelă, unele coloane pot conţine valori duplicate. Adică, pentru mai multe înregistrări, pe acelaşi câmp, vom avea aceeaşi 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 instrucţiunea **SELECT** mai apare un cuvânt cheie, şi 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 reţinut și că această clauză **DISTINCT** poate fi utilzată şi în cadrul funcţiilor de agregare. În acest caz, cuvântul cheie **DISTINCT** este utilizat ca argument al funcţiei de agregare. De exemplu, pentru a număra doar valorile distincte dintr-o coloană.

Un exemplu în acest sens ar fi următoarea instrucţiune **SELECT**, care afişează localităţile de domiciliu ale angajaţilor salvaţi în baza de date a unei companii, în tabela *angajati*. Este evident faptul că, există posibilitatea ca mai mulţi angajaţi să aibă aceeaşi localitate de domiciliu. Deci, pentru a extrage toate localităţile din care avem angajaţi, vom folosi o instrucţiune **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 localităţile care se regăsesc de mai multe ori în tabelă ar fi fost afişate de fiecare dată.

Următoarea instrucţiune va returna toate localităţile de domiciliu, la fel ca mai sus, dar va returna şi judeţul pentru fiecare localitate în parte (în acest caz vor exista judeţe care se repetă):

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

Iată şi un exemplu de folosire a clauzei **DISTINCT** ca argument într-o funcţie de agregare. De exemplu, dacă într-o tabelă în care sunt salvate spre evidenţă facturile unor clienţi ai unei companii, vrem să știm câţi clienţi au facturi emise de companie, avem nevoie de folosirea acestui argument, **DISTINCT**, în cadrul funcţiei **COUNT**:

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

Observăm că absenţa clauzei **DISTINCT** din cadrul funcţiei **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 aceluiaşi client, ceea ce este foarte posibil, rezultatul obţinut ar fi fost alterat, adică nu ar fi corespuns cerinţei noastre de a afla numărul de clienţi unici pentru care există facturi emise de către companie.

4.15 Concluzii

n această lecţie am tratat pe larg comenzile aparţinând **L**imbajului de **M**anipulare a **D**atelor (**LMD**), iar accentul a fost pus pe instrucţiunea de regăsire a datelor pentru care au fost precizate şi explicate toate clauzele posibile. În continuare vor fi tratate aspecte legate de operatorii întâlniţi în **MySQL**, o parte din ei au fost prezentaţi şi în cadrul acestei lecţii, precum şi de funcţiile predefinte pe care **MySQL** le pune la dispoziţia utilizatorilor.

Tema sedinta 4

Creaţi o **bază de date** în care să se păstreze evidenţa 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** reţine informaţii de bază despre profesori: **id_profesor**, **nume**, **prenume**, **localitate,** **data angajării** și **salariu**. Tabela **cursuri** conţine informaţii despre cursurile predate: **id_curs**, **denumire**, **durată (număr de lecții)**.

Cerinţe:

1. Scrieți instrucțiunea **SQL** pentru **crearea bazei de date**.
2. Stabiliți tipurile de dată și dimensiunile pentru fiecare câmp al celor două tabele și scrieţi 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. Scrieţi o instrucţiune **SQL** prin care să **afișați profesorii din provincie (câmpuri afișate: nume, prenume, localitate) ordonați alfabetic după nume și prenume**.
9. Scrieţi o instrucţiune **SQL** prin care să **afișați localitățile** existente în tabela **profesori. Nu afișati valorile duplicate** - **(câmp afișat: localitate)**.
10. Scrieţi o instrucţiune **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)**.

Scrieţi o instrucţiune **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 lecţie vom aborda subiectul referitor la **operatori** şi **funcţii predefinite** în **MySQL**. În ceea ce priveşte operatorii, în **MySQL** avem trei tipuri de operatori:

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

Expresiile care apar împreună cu aceşti 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 înmulţire, „**/**” este operatorul pentru împărţire iar „**%**” este operatorul pentru restul împărţirii a două numere (**modulo**). Dacă avem împărţire 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 utilizaţi 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 „**şi logic”**. În **MySQL** pentru „**şi logic**”, mai avem şi 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 şi 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**” şi 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 şi 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 obţinut să fie adevărat.

Evident că, dacă avem ambele expresii false, 0 **OR** 0 = 0.
Deci, operatorul **OR** (**||**) va returna adevărat atunci când cel puţin una dintre expresii este adevărată, iar în cazul în care ambele expresii sunt false şi 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 adevăraţi sau ambii operanzi sunt falşi, 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 **negaţie**. În **MySQL** pentru negarea unei expresii mai avem şi operatorul „**!**”.

Acest operator de negaţie este foarte simplu de folosit şi de înţeles. Practic, dacă avem o expresie adevărată şi o negăm (îi aplicăm acest operator **NOT**) ea devine falsă, şi reciproc,

dacă o expresie este falsă şi 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 şi 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 şir de caractere are o anumită formă: dacă este prefixat respectiv postfixat sau nu de un anumit subşir, dacă acesta conţine un anumit subşir. Important de reţinut este şi faptul că, în **MySQL,** simbolul „_” (**underline**) ţine loc unui singur caracter, în timp ce simbolul „**%**” ţine 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 menţiune importantă este aceea că atunci când se testează anumite expresii (valori, câmpuri ale unei tabele) 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 prezentaţi anterior și anume **IS NULL**, respectiv **IS NOT NULL**.

BETWEEN - testează dacă o valoare se găseşte între 2 valori date; forma este următoarea ***expresie*** **BETWEEN** ***valoare_minimă*** **AND** ***valoare_maximă*****;** deci, **BETWEEN** verifică dacă ***expresie*** se găseşte în intervalul închis cu capetele ***valoare_minimă***, respectiv, ***valoare_maximă***;
Operatorul **BETWEEN** poate fi înlocuit cu operatorii **>=** şi **<=**. 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 **valore_minimă**, repsectiv, **valoare_maximă**.

IN(***val₁, ..., valₙ***) - testează dacă o valoare aparţine unei mulţimi de valori trecută ca argumente între parantezele operatorului **IN**;

NOT IN(***val₁, ..., valₙ***) - testează dacă o valoare nu aparţine mulţimii de valori dată între parantezele operatorului.

Aceşti operatori au mai fost explicaţi la lecţia anterioară la subcapitolul **Operatori folosiţi în clauza WHERE**, unde puteţi 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 $valoare_1$, $valoare_2$, ..., $valoare_n$. Dacă *expresie* este egală cu una din aceste valori atunci operatorul **CASE** va produce rezultatul corespunzător ce urmează după clauza **THEN**, adică unul dintre $rezultat_1$, $rezultat_2$, ..., $rezultat_n$. 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 $expresie_1$, $expresie_2$, ..., $expresie_n$, 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 funcţii simple care prelucrează fiecare înregistrare şi pentru fiecare înregistrare returnează un rezultat şi funcţii de agregare (de grup) care prelucrează un set de înregistrări şi returnează un singur rezultat pentru acel set de înregistrări.

O funcţie primeşte şi *parametri*. *Parametrii* unei funcţii sunt valori pe care o funcţie le primeşte pentru a le prelucra şi a returna un rezultat. Parametrii unei funcţii se mai numesc şi *argumente*.

Putem clasifica funcţiile simple în mai multe tipuri:

- **matematice;**
- **de comparare;**
- **condiţionale;**
- **pentru şiruri de caractere;**
- **pentru date calendaristice.**

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

5.7 Funcții matematice

Funcţiile matematice sunt folosite pentru efectuarea de operaţii matematice. Ele pot fi utlizate fie pentru realizarea de operaţii matematice cu numele coloanelor, fie cu valori specifice.

În continuare vom prezenta câteva dintre cele mai cunoscute funcţii matematice disponibile în **MySQL**. Fiecare funcţie primeşte unul sau mai mulţi parametri care sunt trecuţi între paranteze rotunde – „**()**”. Aceşti parametri pot reprezenta numele unor coloane sau numere. Astfel, avem funcţiile:

- **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 paramentru, rotunjirea se face fără zecimale; dar această funcţie poate primi şi 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 şi 1; această funcţie nu are parametri;
- **SQRT(*n*)** – returnează valoarea rădăcinii pătrate (radicalul) unui număr;
- **MOD(*a*,*b*) –** returnează restul împărţirii lui a la b; acelaşi rezultat se poate obţine şi 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 funcţii de comparare:

LEAST(val1,val2,...) – returnează cel mai mic parametru dintr-o listă de parametri; această funcţie trebuie să aibă cel puţin 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 funcţia **LEAST()** şi această funcţie trebuie să aibă cel puţin 2 parametri;

INTERVAL(n,n1,n2,...) – această funcţie 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 ...** ; funcţia va returna 0 dacă **n<n1,** 1 dacă **n<n2**, deci va returna poziţia pe care este găsită prima valoare mai mare decât primul parametru al funcţiei, considerând că primul parametru cu care se compară această valoare se află pe poziţia 0, al doilea pe poziţia 1; dacă **n** este **NULL** funcţia va returna -1.

5.9 Funcții condiționale

Continuăm prezentarea cu funcţiile condiţionale:

IF(expresie_testată,expr1,expr2) – această funcţie primeşte trei parametri, se verifică valoarea de adevăr a primului argument al funcţiei, **expresie_testată**, dacă valoarea de adevăr este **TRUE (adevărat)** atunci funcţia returnează cel de-al doilea parametru, **expr1**, altfel, adică valoarea de adevăr a evaluării expresiei de testat este **FALSE (fals)** atunci funcţia va returna **expr2**.

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

NULLIF(expr1,expr2) – această funcţie compară cele 2 expresii primite ca parametri, dacă sunt egale funcţia 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 tabelei se pot trece nişte specificatori. Ne vom ocupa aici de specificatorii **NULL** respectiv **NOT NULL**. Primul, care este şi 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 şi 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;
- şirul vid dacă acea coloană este de tip şir 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 menţionat aspectul că nu trebuie să se facă confuzie între valoarea **NULL** şi 0, de exemplu pentru coloane numerice, sau şirul vid (**""**) pentru coloane de tip şir de caractere. Acestea sunt valori, şi 0 şi şirul vid sunt valori, pe când **NULL** specfică de fapt **lipsa unei valori** în acel câmp.

5.11 Funcții pentru şiruri de caractere

Funcţiile pentru şiruri de caractere sunt folosite atunci când se lucrează cu şiruri de caractere, deci parametrii acestor funcţii pot fi coloane sau valori de tip şir de caractere. Astfel, prezentăm în continuare câteva dintre aceste funcţii pentru text:

CONCAT(s1,s2,s3,...) – va concatena (alipi) toate şirurile date ca argumente ale funcţiei; dacă unul dintre argumente este **NULL** atunci rezultatul funcţiei va fi **NULL**.
CONCAT_WS(separator,s1,s2,s3,...) – va concatena (alipi) toate şirurile date ca argumente ale funcţiei cu separatorul dat ca prim argument între ele; dacă separatorul este **NULL** atunci rezultatul funcţiei va fi **NULL**, dar dacă unul din celelalte argumente este **NULL,** atunci funcţia va returna celelate şiruri concatenate, ignorând **NULL**. **Atenţie**, în **MySQL NU** există un operator de concatenare (ca în PHP, spre exemplu). Deci, în MySQL, pentru concatenare vom folosi una din aceste funcţii.
CHAR_LENGTH(s) – returnează lungimea șirului *s*;
LENGTH(s) - lungimea (nr. de caractere) şirului *s*;
LPAD(s,lungime_finală,şir_completare) – șirul *s* este completat la stânga cu șirul *șir_completare* până ajunge la lungimea *lungime_finală*;
RPAD(s,lungime_finală,şir_completare) – șirul *s* este completat la dreapta cu șirul *șir_completare* până ajunge la lungimea *lungime_finală*;
LTRIM(s) - întoarce şirul obţinut din *s* prin eliminarea spaţiilor inutile din stânga;
RTRIM(s) - întoarce şirul obţinut din *s* prin eliminarea spaţiilor inutile din dreapta;
TRIM(s) - întoarce şirul obţinut din *s* prin eliminarea spaţiilor inutile atât din dreapta cât şi 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 şirul obţinut din *s* prin convertirea tuturor literelor mici la litere mari;
LOWER(s) - întoarce şirul obţinut 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 funcţii **MySQL** este cea a funcţiilor 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 şi ora este *year-month-day hour-minute-second* (AAAA-LL-ZZ HH-MM-SS sau YYYY-MM-DD HH-MM-SS).

Astfel avem următoarele funcţii 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 şi ora curentă;
- **NOW()** – returnează data şi 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 funcţiei (intervalul de valori este de la 0 la 23 dacă parametrul reprezintă o oră dar poate lua şi valori mai mari dacă parametrul reprezintă un timp contorizat – număr de ore, minute şi secunde contorizate);
- **MINUTE(time)** - returnează minutul din valoarea parametrului funcţiei (intervalul de valori care poate fi returnat este de la 0 la 59);
- **SECOND(time) -** returnează seunda din valoarea parametrului funcţiei (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 funcţii care operează asupra mai multor înregistrări (linii) dintr-o tabelă a unei baze de date, calculând şi returnând o singură valoare. Aceste funcţii poartă denumirea de funcţii agregat sau funcţii de agregare. Avem funcţii de agregare matematice şi o funcţie agregat pentru numărare.

Funcţiile 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 şi funcţia de agregare de numărare. Este vorba de funcţia:

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

Aceste funcţii de agregare au fost prezentate pe larg, cu exemple concrete de utilizare, în cadrul lecţiei numărul 4 la prezentarea clauzelor instrucţiunii **SELECT.** În instrucţiunea de regăsire a datelor, instrucţiunea **SELECT**, pentru a utiliza funcţiile de agregare trebuie folosită clauza **HAVING**, deci, funcţiile 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 celelate 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 lecţii am prezentat operatorii întâlniţi în **MySQL** şi câteva dintre cele mai cunoscute funcţii predefinte ale acestui **SGBD**.

În lecţia următoare ne vom ocupa de **uniunile** între tabele sau **join**-uri, tipuri de **join**-uri, precum şi de reuniuni. Aşadar, vom trece la noţiuni 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 noţiunea de uniune de tabele. Întrucât ne ocupăm de baze de date relaţionale, ştim că tabelele din baza de date sunt legate între ele (sunt relaţionate). Nu este indicat să avem tabele izolate într-o bază de date. Pe parcursul acestui curs am făcut referire în mai multe rânduri la relaţionarea dintre tabelele unei baze de date. De asemenea, au fost prezentate şi tipurile de relaţii posibile între tabele.

Aşa 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 şi 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 situaţii în care este necesară extragerea informaţiilor din mai multe tabele. Pentru acest lucru trebuie realizată o relaţie între tabele pe baza unor informaţii comune. Această interogare a bazei de date în care se realizează relaţii între tabele pe baza unor informaţii comune (câmpuri comune, coloane comune) se numeşte **joncţiune** (**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_nume**, introduse în clauza **FROM** a instrucţiunii **SELECT**. **Alias**-urile pot fi utilizate în orice parte a instrucţiunii **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 tabelelor. Un alt avantaj al **alias**-urilor este că putem realiza auto-uniuni, în cadrul cărora aceeaşi tabelă poate avea **alias**-uri diferite. Acest aspect l-am explicat şi când am prezentat **SELF JOIN**. Utilizarea unui **alias** duce şi la o scurtare a sintaxei **SQL** într-o comandă.

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

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

6.3 Tipuri de join-uri

Există mai multe *tipuri de join-uri*. O clasficare 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 selecţie în care se trec toate câmpurile pe care dorim să le obţinem, din cadrul ambelor tabele, iar la clauza **FROM** se trec ambele tabele, fiecare dintre ele putând avea definit şi un **alias**, adică un nume mai scurt prin care pot fi accesate.

Nu există o limită în ceea ce priveşte numărul de tabele ce pot fi unite în cadrul unei instrucţiuni **SELECT**. Aşadar, putem avea două sau oricât de multe tabele unite printr-o **joncţiune**. 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 egaliate între coloanele comune ale tabelelor 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 jocțiuni *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 gestiune 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 tabelelor** 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** intre tabelele relaționate și se subînțelege că avemde-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 tabelei 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 scriere*a în cazul în care acea 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 specifcăm, 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 mulţimilor, considerând cele două tabele ca fiind două mulţimi **A** şi **B**, **LEFT JOIN** între cele două tabele (**A** şi **B**) reprezintă **(A \ B) U (A ∩ B)**. Deci, **LEFT JOIN** reprezintă rezultatul diferenţei dintre cele două mulţimi reunit cu rezultatul intersecţiei celor două mulţimi.

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 acelaşi rezultat, cu deosebirea că se vor extrage înregistrările din a doua tabelă care au sau nu înregistrări în prima. Este, practic, reciproca joncţiunii **LEFT JOIN**.

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

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

*Este valabilă şi 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 tabelei 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 aceia 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 tabele la realizarea asocierii de tabele* (joncțiunii**) prezența alias-urilor pentru tabele 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 tabele cu ea însăși este cea prezentată la **LEFT**, respectiv, **RIGHT JOIN**, deosebirea fiind dată de daptul că ambele mulțimi (tabele) participante la joncțiune au același nume (pentru că e vorba de aceeași tabelă) dar alias-uri diferite), deosebirea constă în faptul că ambele mulțimi (tabele) participante la realizarea asocierii sunt de fapt una și aceași, 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 conţin o singură instrucţiune **SELECT**, care returnează date din una sau mai multe tabele. Dar este permisă şi efectuarea mai multor interogări şi 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 instrucţiuni **SELECT**, separate prin cuvântul cheie **UNION** sau **UNION ALL**, iar fiecare instrucţiune **SELECT** trebuie să conţină acelaşi număr de coloane, expresii sau funcţii agregat, dar nu contează ordinea în care sunt specificate coloanele. Coloanele din instrucţiunile **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 instrucţiuni **SELECT** dintr-o reuniune, este afişată o singură dată această înregistrare.

Sintaxa unei instrucţiuni 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 instrucţiuni **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 obţinute în urma efectuării unei reuniuni, clauza **ORDER BY** se trece la final, după ultima instrucţiune **SELECT**.

Reluăm sintaxa instrucţiunii de reuniune de interogări, adăugând şi 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 lecţie am abordat noţiunile de **joncţiune** sau **uniune de tabele** şi **reuniune**. Am prezentat tipurile de joncţiuni (**join**-uri) care se pot întâlni în **SQL**, şi, de asemenea am discutat despre reuniunea interogărilor folosind operatorul **UNION,** respectiv**, UNION ALL**.

Aşadar, începând cu această lecţie am început să expunem lucruri mai complexe din domeniul bazelor de date. Lecţia următoare continuă să abordeze teme cu o complexitate mai mare, şi 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 instrucţiunea **SELECT** din interior (**subinterogarea**) şi apoi interogarea exterioară.

Fiecare interogare conţine 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 obţinerea de câmpuri cu valoare calculată, cum ar fi numărul de înregistrări care îndeplinesc o anumită condiţie dintr-o altă tabelă decât cea din care se extrag celelate câmpuri în instrucţiunea **SELECT** principală. În acest caz, în instrucţiunea **SELECT** de tip **subinterogare** trebuie prefixate numele câmpurilor cu numele tabelei pentru a nu exista neclarităţi.

Rezultatul obţinut din **subinterogare** va primi un nume care poartă denumirea de **alias** care va fi folosit în instrucţiunea 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 şi termenul în limbă engleză ce defineşte o astfel de instrucţiune, este vorba despre termenul **query**.

Evident, în mod similar, pentru o **subinterogare** se poate întâlni frecvent şi denumirea în limba engleză, şi anume, **subquery**.

În cadrul interogării principale, pentru introducerea unei subinterogări se foloseşte un operator de comparaţie. Deci, se compară rezultatul returnat de subinterogare cu un rezultat obţinut 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 conţine 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 instrucţiuni **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 aceași frază **SQL**.* Numărul de interogări imbricate depinde și de dimeniunea 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 rezultele, 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 menţionată într-o subinterogare, dar nu va influența rezultatele afișate de interogarea principală, deoarece rezultatele subinterogării nu sunt afişate, 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 principlaă. Pot fi ordonate, în schimb, și rezultatul va fi vizibil la afișare, înregistrările returnate de interogarea principală **SELECT**, care au prelucrat şi 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 principlae**)**, cele mai utilizate, a instrucţiunii **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 egalităţii sau inegalităţii unor valori, al apartenenţei 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** şi **un singur rând** dintr-o tabelă. Condiţia este pusă pe cheia primară.

Interogarea care întoarce valoarea se scrie inclusă între paranteze rotunde „**()**”, din acest moment ea se comportă ca şi 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 **înşiruire 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 apartenenţa 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 foloseşte 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 foloseşte cuvântul cheie **EXISTS** care poate fi şi negat, deci, se poate folosi şi 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 existenţa 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 tabelei care va prefixa fiecare utilizare a unei coloane extrase* în cadrul subinterogării.

Alias-ul se specifică tot cu ajutorul clauzei **AS**, aşa cum am expus în prezentarea **alias-**urilor descrisă în lecţiile anterioare. Trebuie, de asemenea, inclusă între paranteze rotunde „**()**”. Câmpurile tabelei întoarse de subinterogare au numele exact ca în antetul care s-ar afişa 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 tabelei, urmat de caracterul „**.**” şi 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** şi **ALL** se folosesc în combinaţie cu operatorii de comparare**.** Operatorul **ANY** înseamnă oricare element din listă, în timp ce **ALL** înseamnă toate elementele din listă. Operatorii **ANY** şi **ALL** sunt operatori de cuantificare, se mai numesc şi cuantificatori. Aceştia 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, condiţia 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, condiţia 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 lecţie am discutat despre noţiunea de **subinterogare** și despre **tipurile de subinterogări**. Sunt noţiuni complexe în lucrul cu baze de date, însă utile în foarte multe situaţii.

Următoarea lecţie 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 ia 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, nişte **tabele virtuale**, adică vederile conţin interogări care regăsesc în mod dinamic datele atunci când sunt utilizate. Ele nu conţin 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 operaţiunilor **SQL** complexe. După ce a fost scrisă o interogare aceasta poate fi refolosită cu uşurinţă dacă a fost creată o vedere pe baza acelei interogări. În plus, rezultatele din tabela virtuală se actualizează dinamic, în funcţie de ceea ce se întâmplă în tabelele din care au fost extrase datele în interogarea care a fost declarată la crearea tabelei virtuale.

Vederile sau **tabelele virtuale** sunt adesea denumite cu termenul în limba engleză, este vorba de termenul **view**. Mai trebuie precizat şi faptul că,**vederile** (**view**-urile) pot fi folosite pentru a folosi doar părţi din anumite tabele, iar nu tabele complete.

După ce au fost create **vederile**, se pot efectua operaţii de interogare (instrucţiuni **SELECT**) şi pe acestea. Ca şi tabelele, vederile trebuie să aibă un nume unic în cadrul unei baze de date. O **vedere nu** poate primi **numele unei tabele** 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 instrucţiune **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 părţi din tabelă vizibile anumitor utilizatori.

Vederile pot fi actualizabile sau neactualizabile. Cele actualizabile permit instrucţiunile **INSERT**, **UPDATE**, **DELETE**. Pe cele neactualizabile nu se pot efectua aceste operaţii. Pentru a fi actualizabile trebuie să fie o corespondenţă linie cu linie între o tabelă virtuală şi tabela pe baza căreia a fost creată.

Este evident acest lucru, dacă nu există această corespondenţă între view şi tabelă nu se pot face actualizări în tabela virtuală.

De asemenea, este important de menţionat că nu trebuie să avem în instrucţiunea care creează tabela virtuală actualizabilă **funcţii de grup**, clauzele **DISTINCT**, **GROUP BY** şi **HAVING**. Nu putem avea nici**subinterogări**, **reuniuni** sau **OUTER JOIN**. În schimb, dacă avem **INNER JOIN** în instrucţiunea de creare, atunci este permisă actualizarea.

8.1.1 Crearea unei tabele virtuale

Sintaxa folosită pentru a crea o vedere (view) este următoarea:

```
CREATE VIEW nume_vedere AS
        SELECT ....;
```

8.1.2 Redefinirea unei tabele virtuale

O **vedere** (**view**) poate fi redefinită. În acest sens, avem o instrucţiune specială, similară cu cea de creare a tabelei virtuale.

Sintaxa folosită pentru **redefinirea** unui **view** este următoarea:

```
ALTER VIEW nume_vedere AS
        SELECT ....;
```

8.1.3 Ștergerea unei tabele virtuale

Evident, avem şi posibilitatea de a şterge o tabelă virtuală din baza de date. Instrucţiunea care permite ştergerea unei vederi din baza de date este următoarea:

```
DROP VIEW nume_vedere;
```

Alte menţiuni pe care le vom face în legătură cu vederile se referă la faptul că o vedere poate fi creată şi pe baza unei alte vederi, deci, nu este obligatoriu să fie realizată pe baza unei tabele. Tabelele virtuale şi tabelele din baza de date sunt stocate în acelaşi 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 tranzacţii 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 temproare 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 triggere (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 permanent 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 tabelei temporare. Aceasta

însemnă 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 tabelei, 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 tabelei.

Recomandarea este să fie utilizata instrucțiunea **DROP TEMPORARY TABLE** urmată de numele tabelei 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 lămuri 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**CREATE TEMPORARY TABLE**	Sunt create prin instrucțiunea**CREATE VIEW**
Conțin efectiv date atât timp cât sunt create și sesiunea curentă este activă	Nu conțin date efectiv ci fa 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 aceeați 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, *creează 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 interogăriii 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 autmată 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 celelate 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 tabelei, 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 ineficientă 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 înregitrări;
- 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 ineficientă 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 ineficientă 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 oa numită 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 tabelei, deci, în cadrul instrucțiunii**CREATE TABLE** sau, ulterior, prin executarea unei instrucțiuni **ALTER TABLE** sau **CREATE INDEX**.

La crearea tabelei, în cadrul sintaxei **CREATE TABLE** se adaugă clauze suplimentare: **PRIMARY KEY**, **UNIQUE** sau **INDEX**. Clauza se adauagă 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_coloană$_1$ [**ASC**|**DESC**][, nume_coloană$_2$ [**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 tabele se poate face prin executarea comenzilor:

- **SHOW CREATE TABLE** – ce va afișa indecșii ca parte a definiției tabelei;
- **DESCRIBE** – care afișează structura unei tabele, 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**.

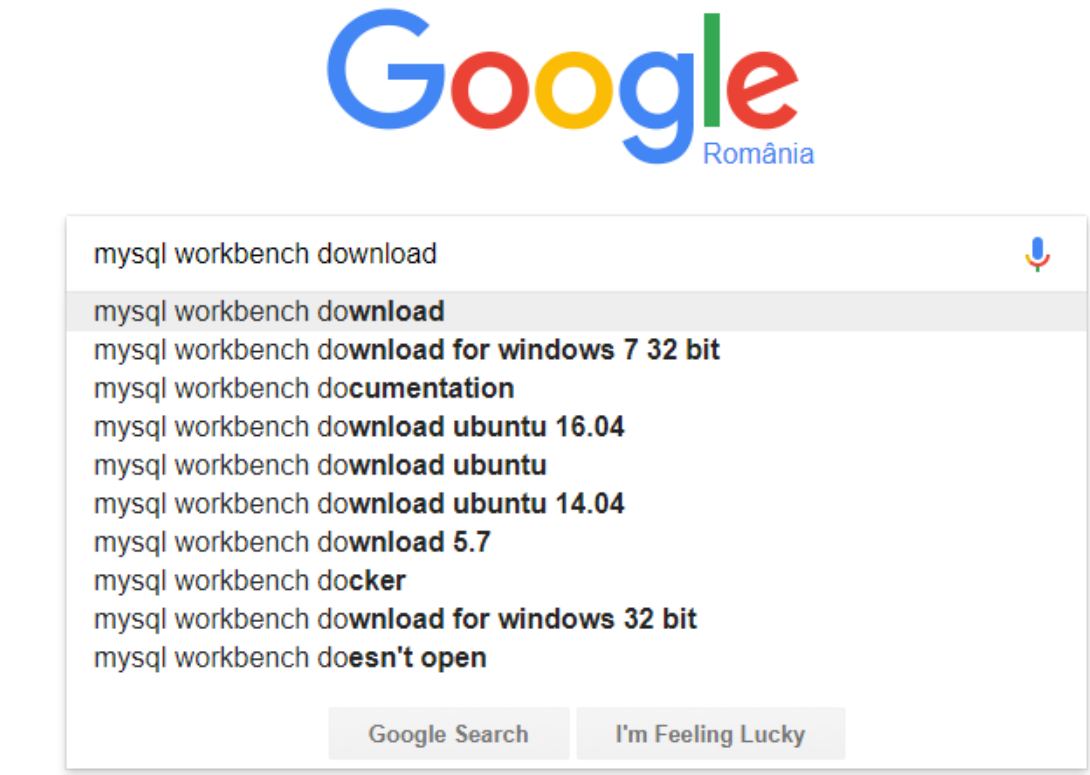
Eliminarea unei constrângeri care este și index, prin executarea unei instrucțiuni **ALTER TABLE,** se face prin comanda următoarea:

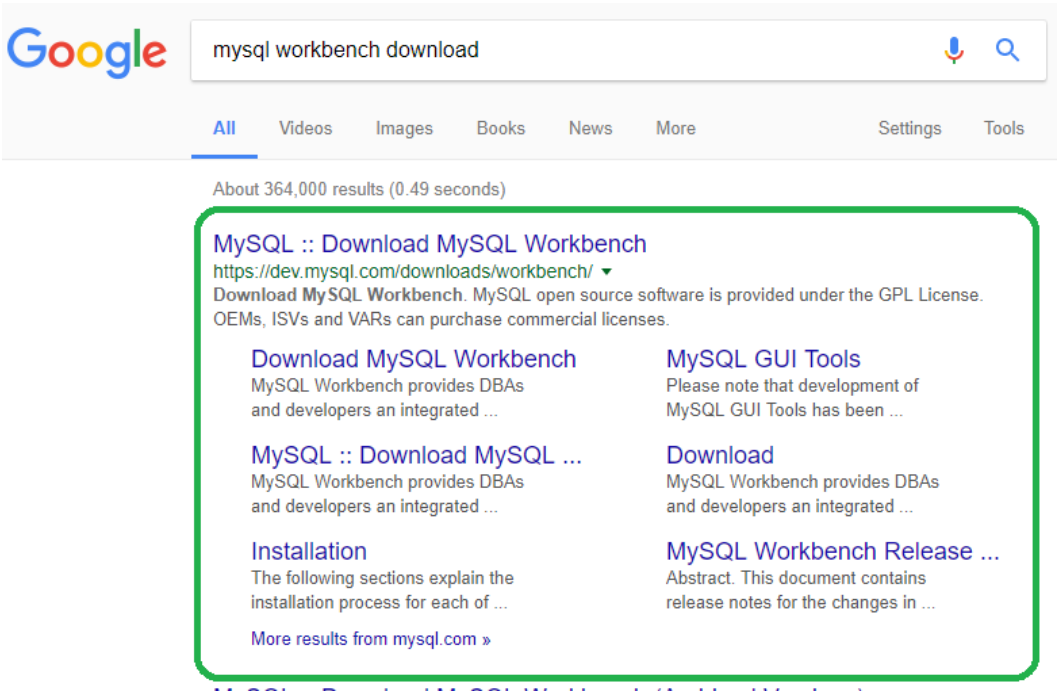
ALTER TABLE *nume_tabelă* **DROP** tip_index [(nume_index)];
Instrucțiunea de ștergere a unui index simplu este următoarea:
DROP INDEX *nume_index* **ON** *nume_tabelă*;

8.5 Concluzii

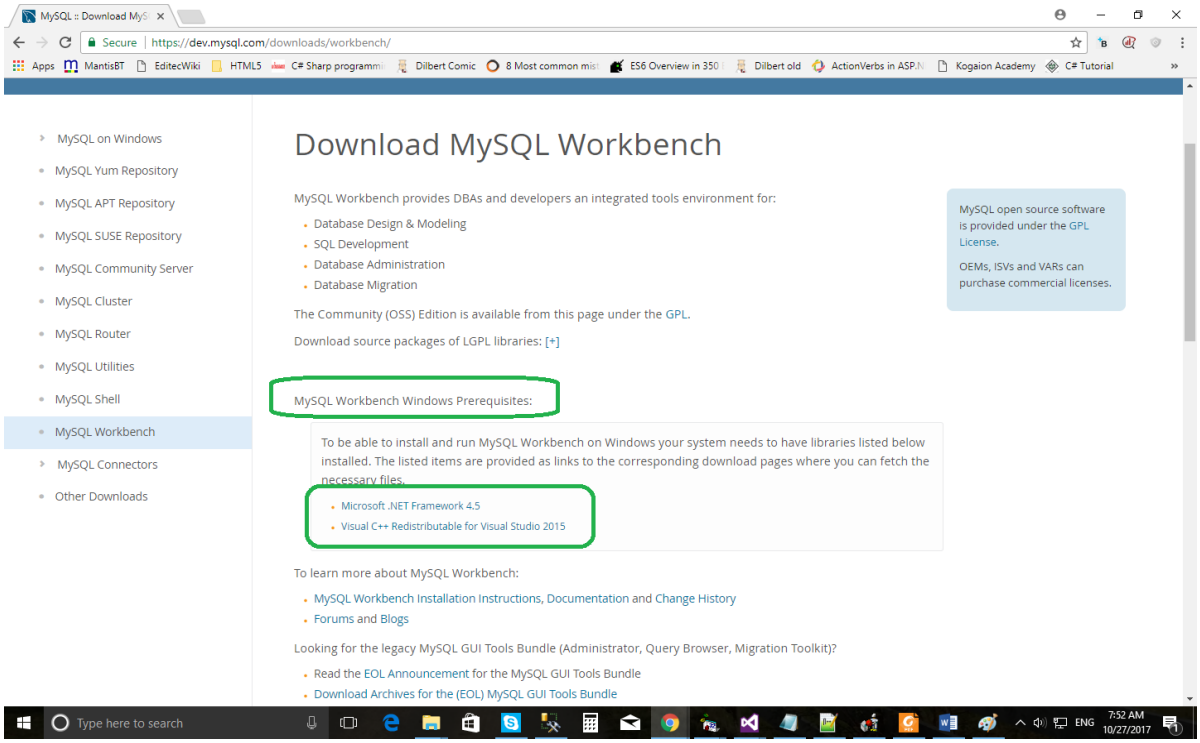
În concluzie*, indexarea este o metodă deosebit de eficientă pentru creșterea eficienței execuției instrucțiunilor de regăsire a datelor (**SELECT**)*, dar, în același timp, poate duce la *îngreunarea execuției instrucțiunilor de manipulare a datelor ce afectează datele din tabelele unei baze de date* (este vorba de instrucțiunile **INSERT**, **UPDATE** și **DELETE**).
Indexarea nu schimbă ordinea fizică a înregistrărilor din tabele. *Prin definirea indecșilor se crează automat o corespondență între ordinea fizică a înregitrărilor și ordinea logică*. Indecșii ordonează la nivel logic după anumite criterii stabilite la definirea (crearea) lor.

Instalare My sql WorkBrench

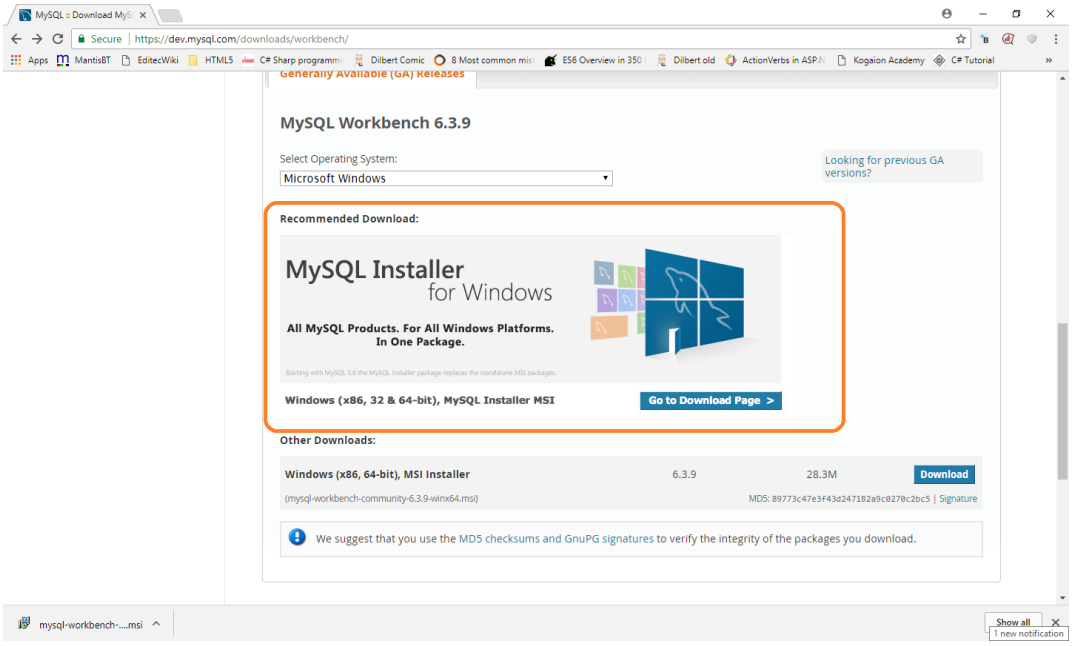


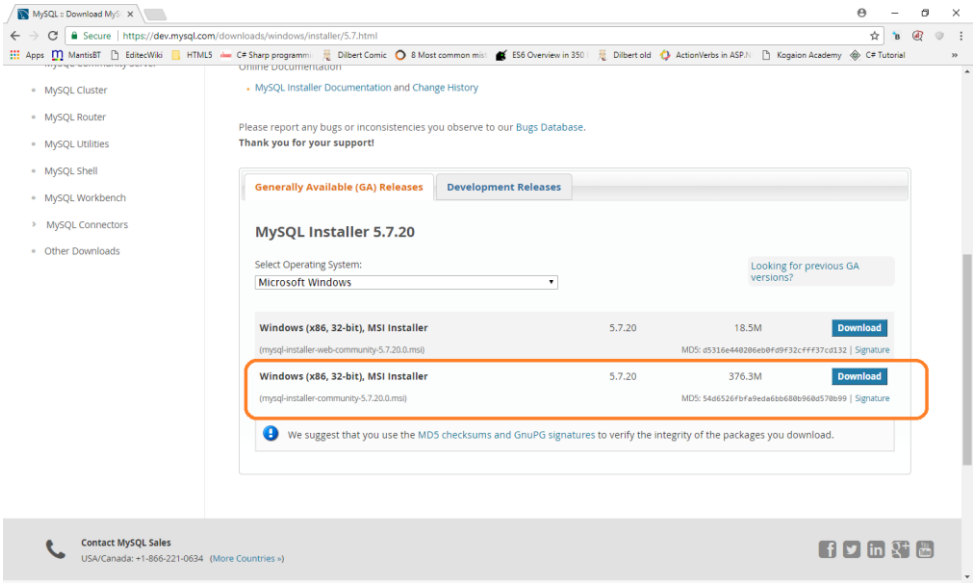


Instalati .Net Framework 4.5 si Visual C++ 2015 – daca este nevoie.

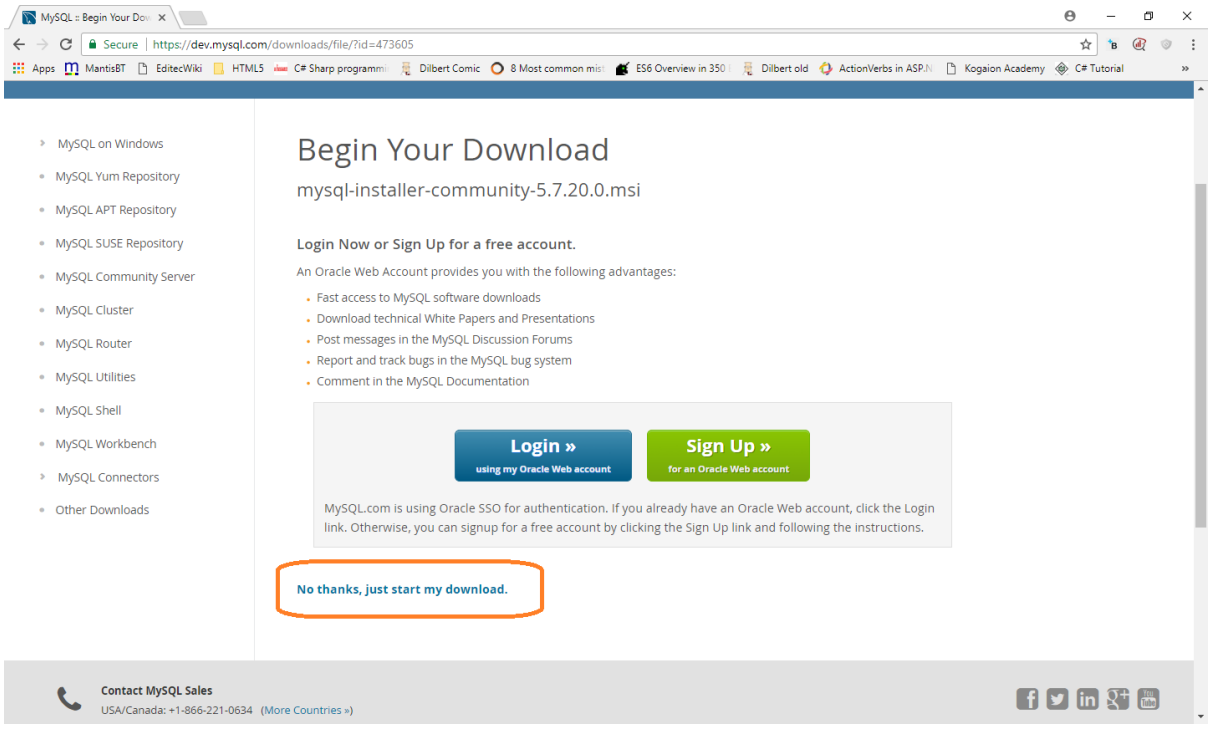


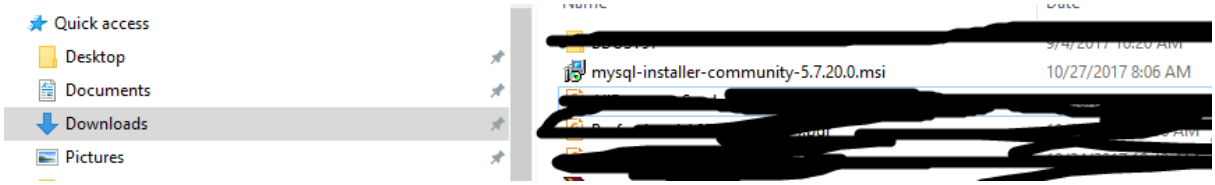
Dupa, in josul acestei pagini gasiti linkurile de instalare:

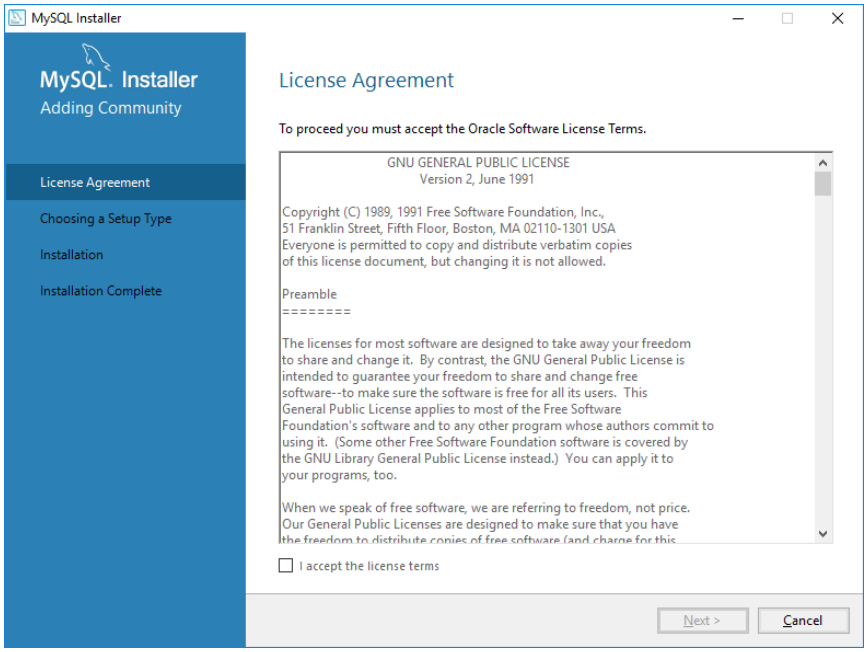




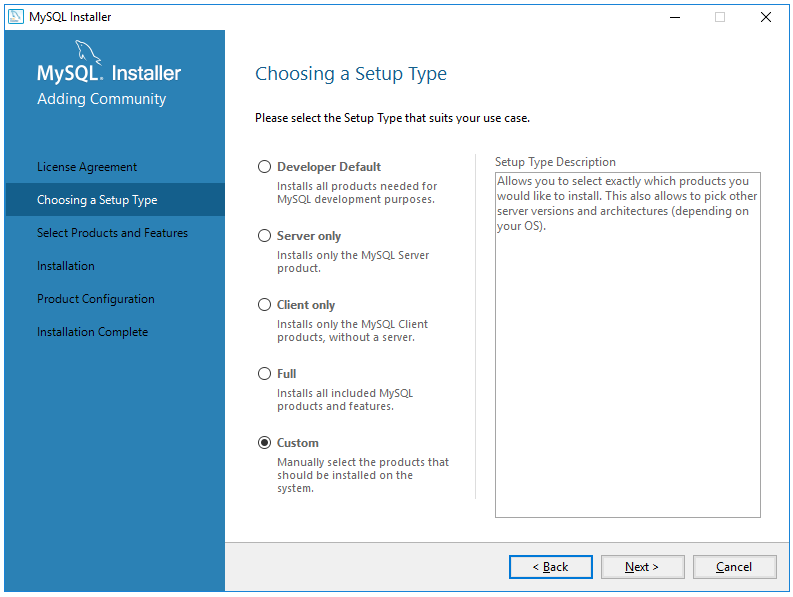
Alegeti pe cel mare la dimensiuni (376 M) – asta inseamna ca are si serverul in kitul de instalare.



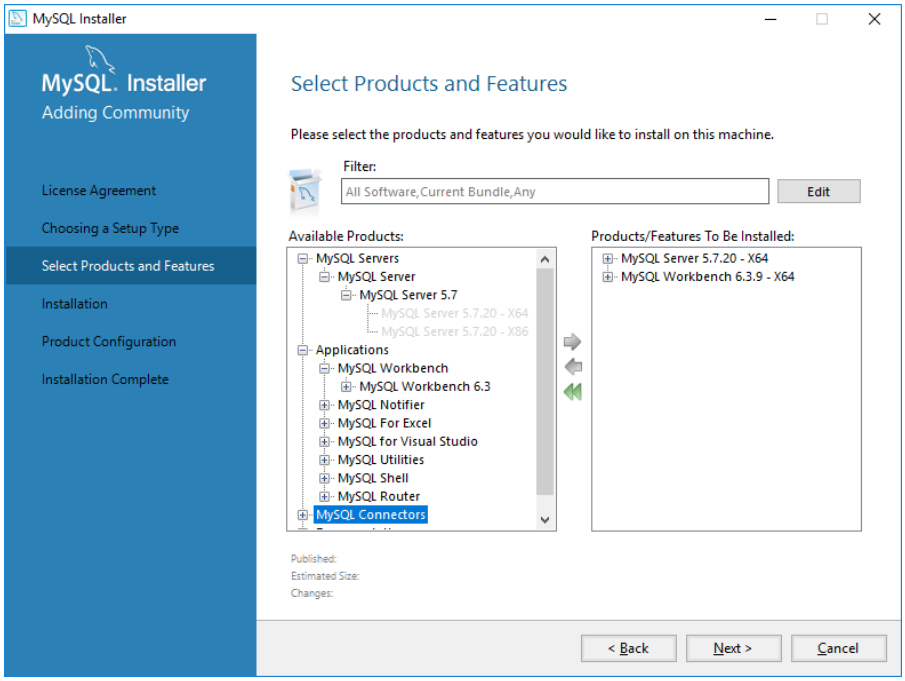




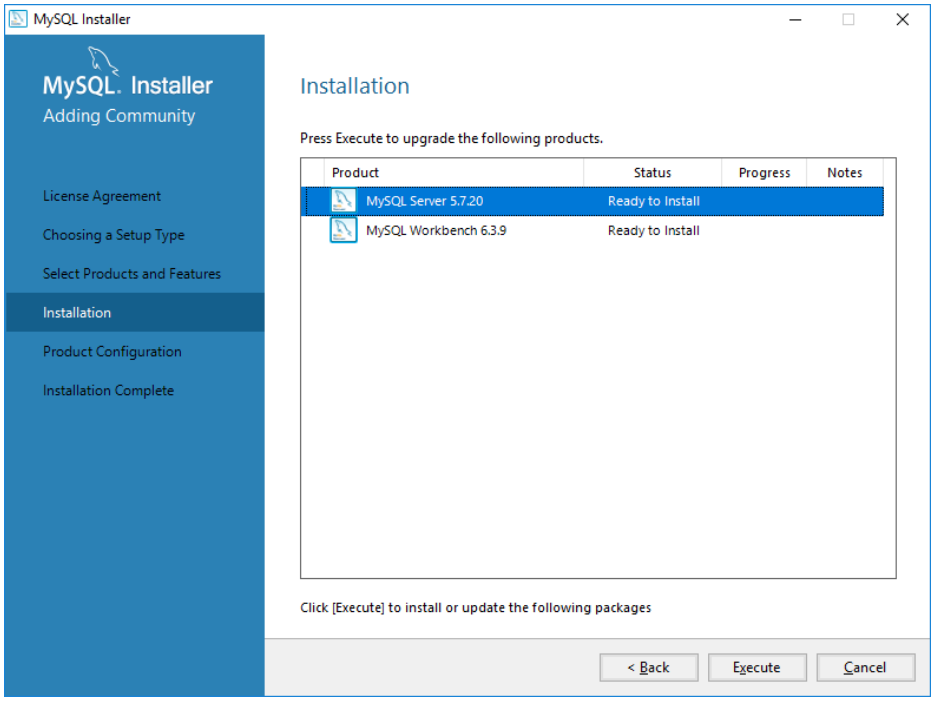
Checked “I accepted the license terms”.



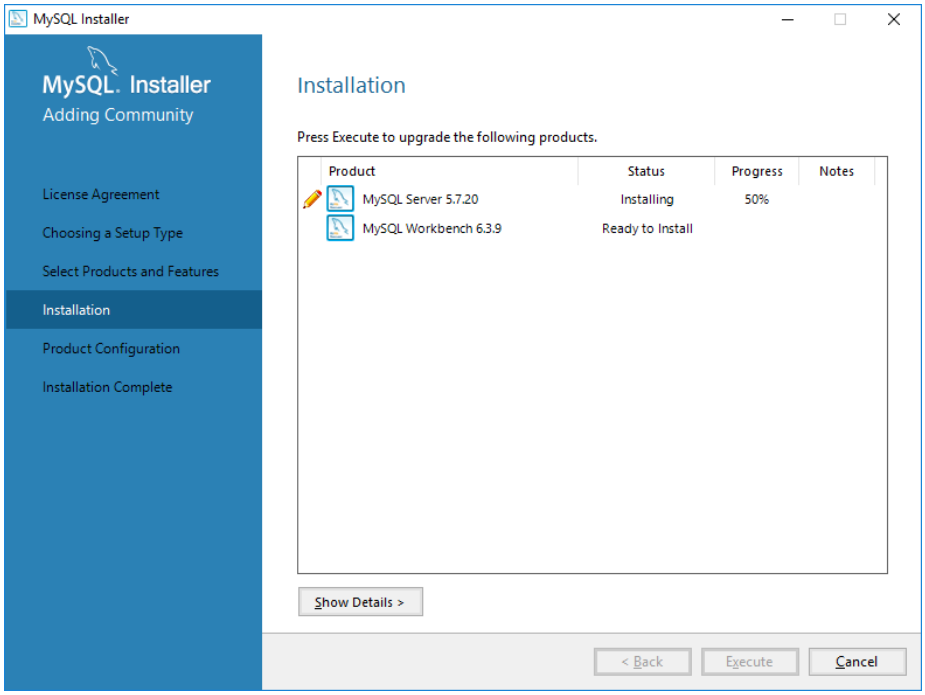
Alegeti de tip “Custom” – nu este necesara instalarea a tuturor produselor

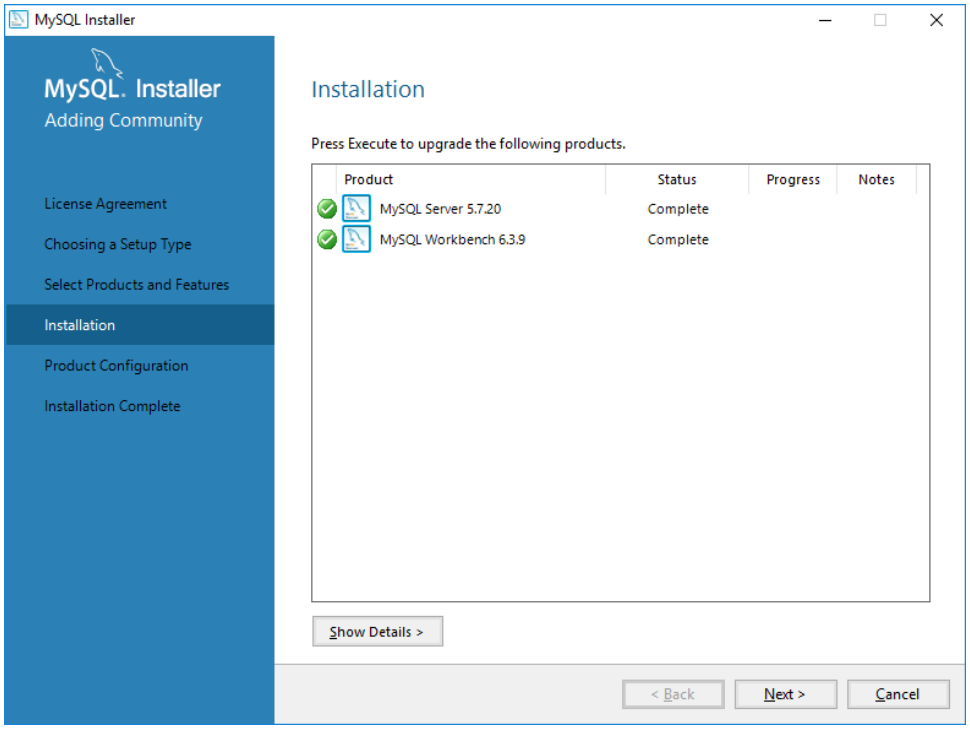


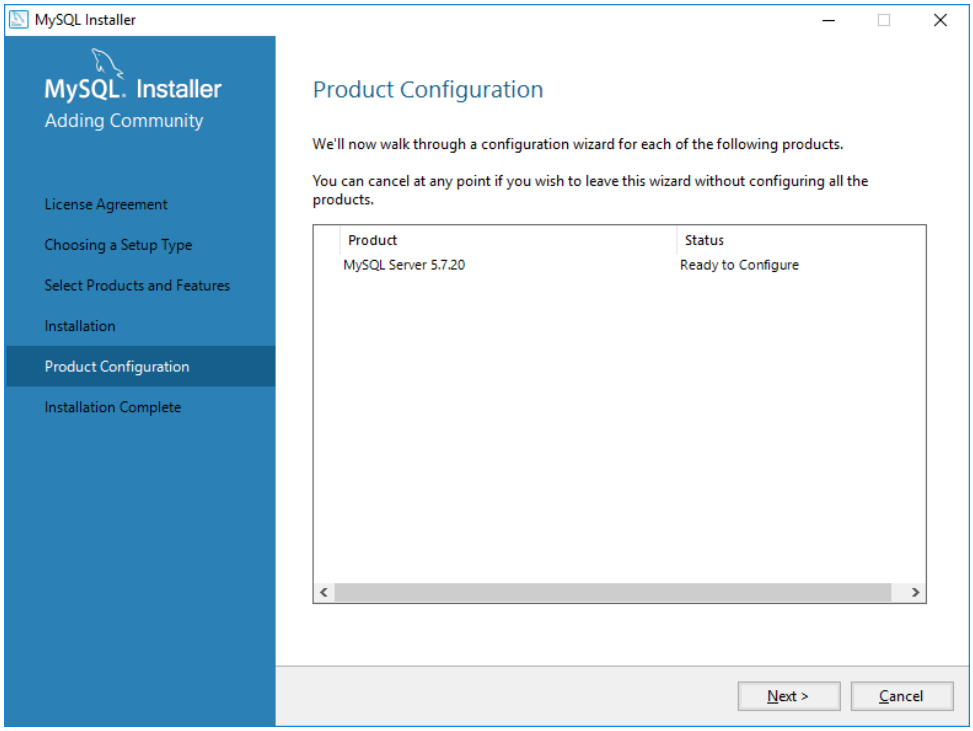
Alegeti MySQL Server si MySQL WorkBench

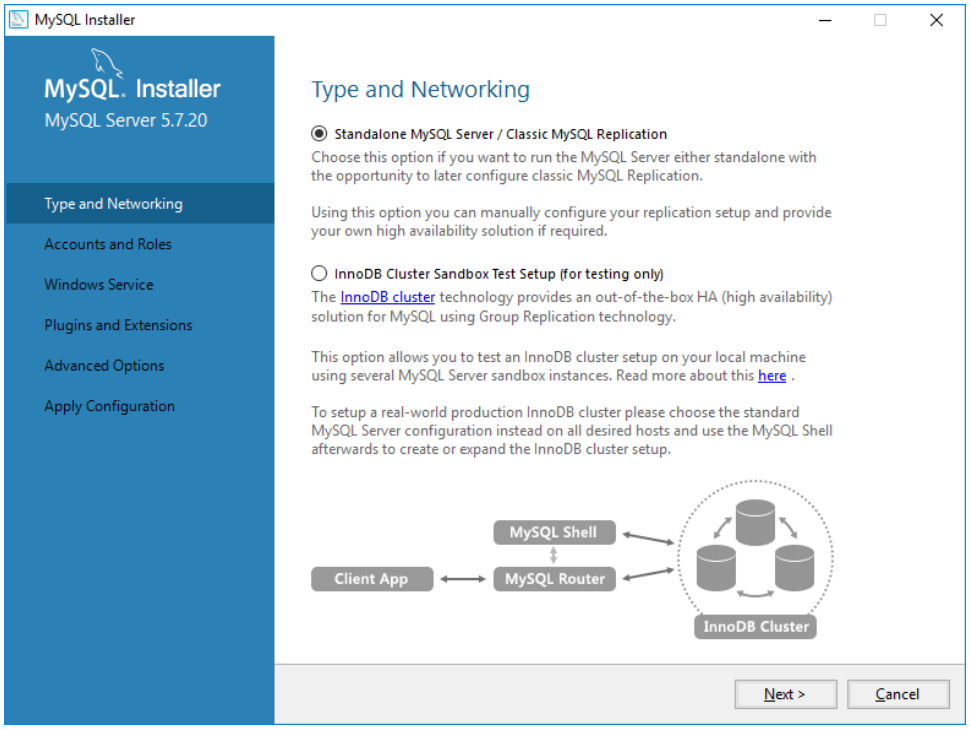


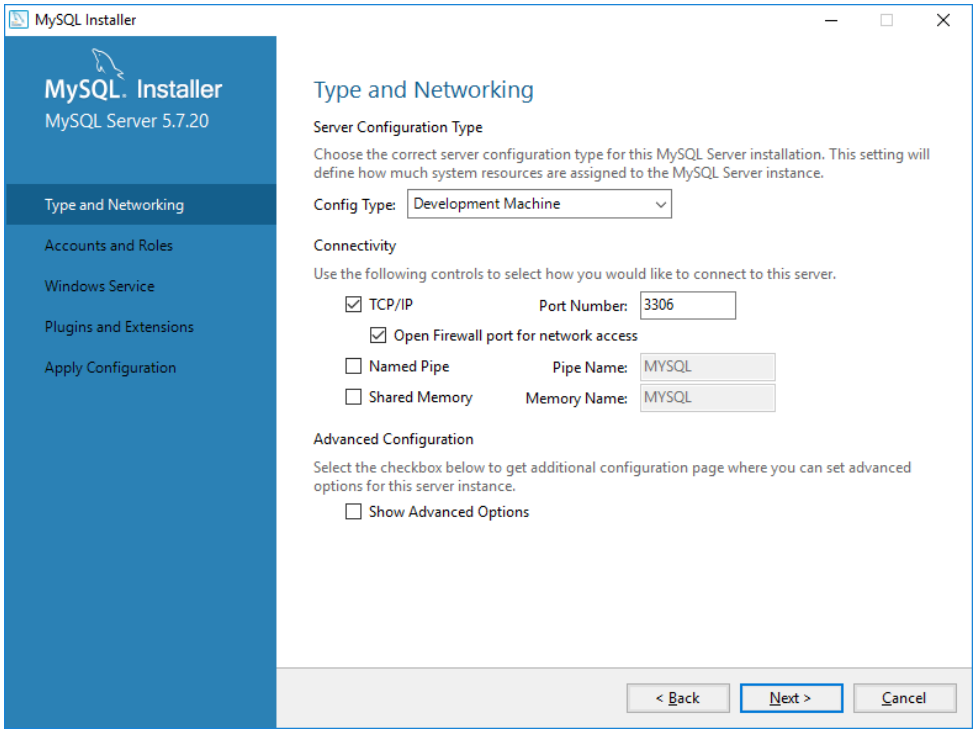
Click pe butonul “Execute”


MySQL Installer
MySQL. Installer
Adding Community
License Agreement
Choosing a Setup Type
Select Products and Features
Installation
Product Configuration
Installation Complete
Installation
Press Execute to upgrade the following products.
Product
Status
Progress
Notes
MySQL Server 5.7.20
Installing
50%
MySQL Workbench 6.3.9
Ready to Install
Show Details >
< Back
Execute
Cancel

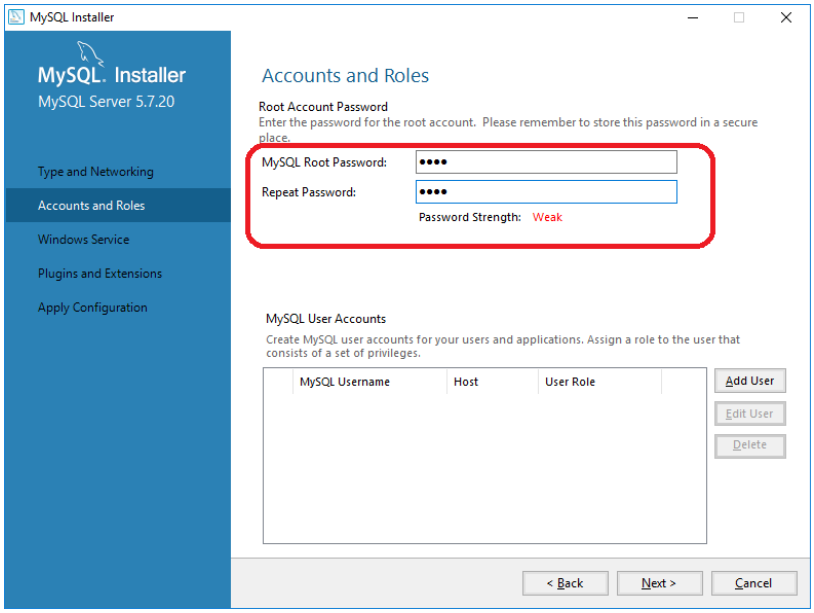

MySQL Installer
MySQL. Installer
Adding Community
License Agreement
Choosing a Setup Type
Select Products and Features
Installation
Product Configuration
Installation Complete
Installation
Press Execute to upgrade the following products.
Product
Status
Progress
Notes
MySQL Server 5.7.20
Complete
MySQL Workbench 6.3.9
Complete
Show Details >
< Back
Next >
Cancel

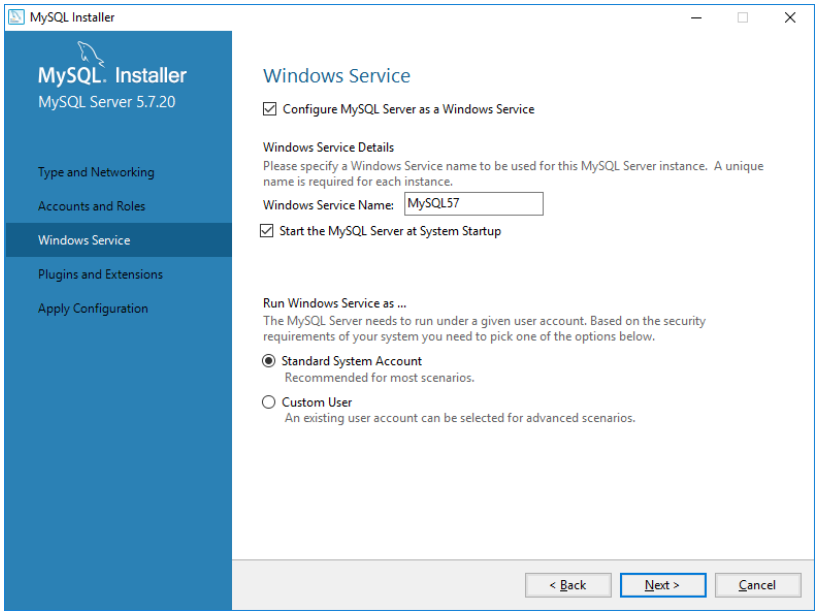

MySQL Installer
MySQL. Installer
Adding Community
License Agreement
Choosing a Setup Type
Select Products and Features
Installation
Product Configuration
Installation Complete
Product Configuration
We'll now walk through a configuration wizard for each of the following products.
You can cancel at any point if you wish to leave this wizard without configuring all the products.
Product
Status
MySQL Server 5.7.20
Ready to Configure
Next >
Cancel

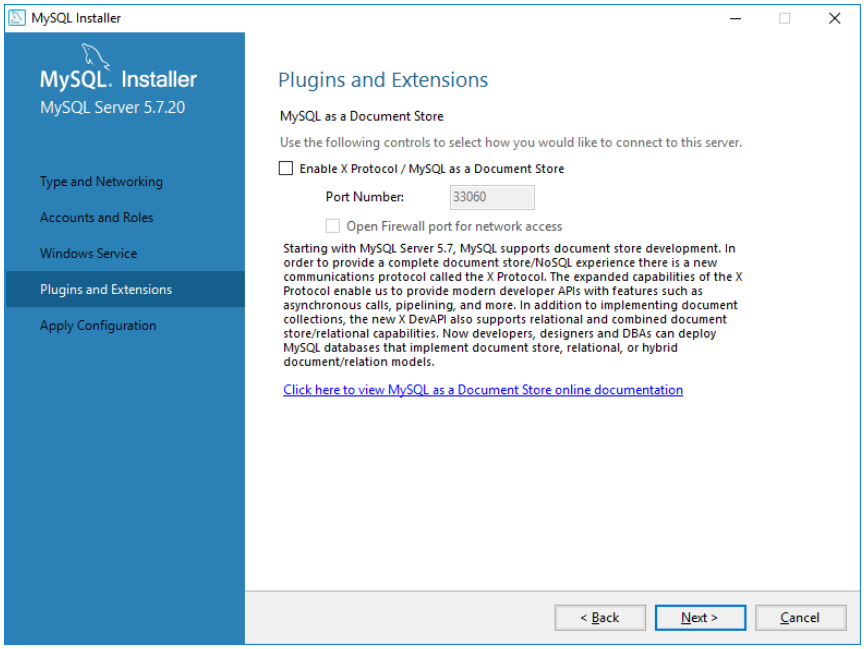

MySQL Installer
MySQL. Installer
MySQL Server 5.7.20
Type and Networking
Accounts and Roles
Windows Service
Plugins and Extensions
Advanced Options
Apply Configuration
Type and Networking
Standalone MySQL Server / Classic MySQL Replication
Choose this option if you want to run the MySQL Server either standalone with the opportunity to later configure classic MySQL Replication.
Using this option you can manually configure your replication setup and provide your own high availability solution if required.
InnoDB Cluster Sandbox Test Setup (for testing only)
The InnoDB cluster technology provides an out-of-the-box HA (high availability) solution for MySQL using Group Replication technology.
This option allows you to test an InnoDB cluster setup on your local machine using several MySQL Server sandbox instances. Read more about this here .
To setup a real-world production InnoDB cluster please choose the standard MySQL Server configuration instead on all desired hosts and use the MySQL Shell afterwards to create or expand the InnoDB cluster setup.
MySQL Shell
Client App
MySQL Router
InnoDB Cluster
Next >
Cancel

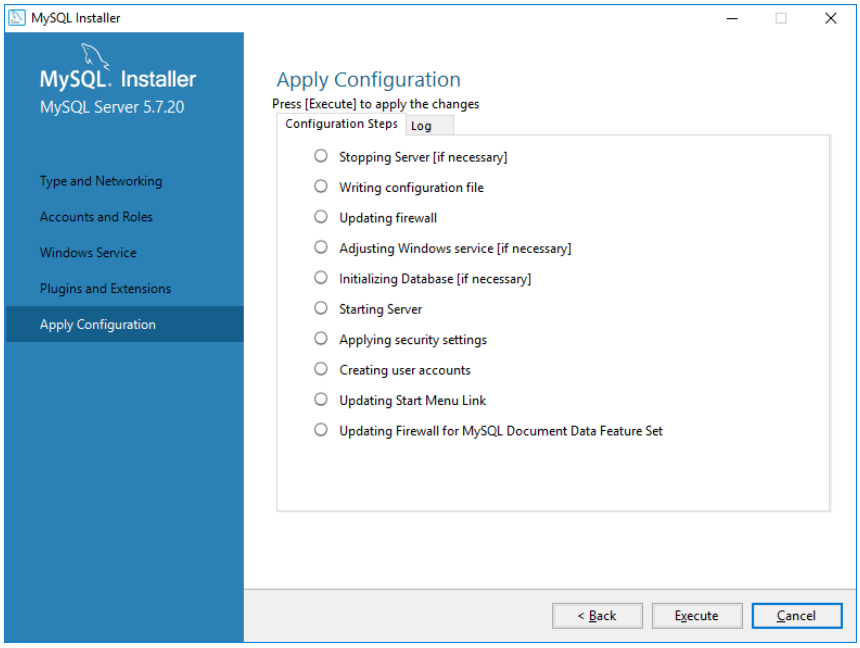


Alegeti o parola simpla, de exemplu eu am ales “root”

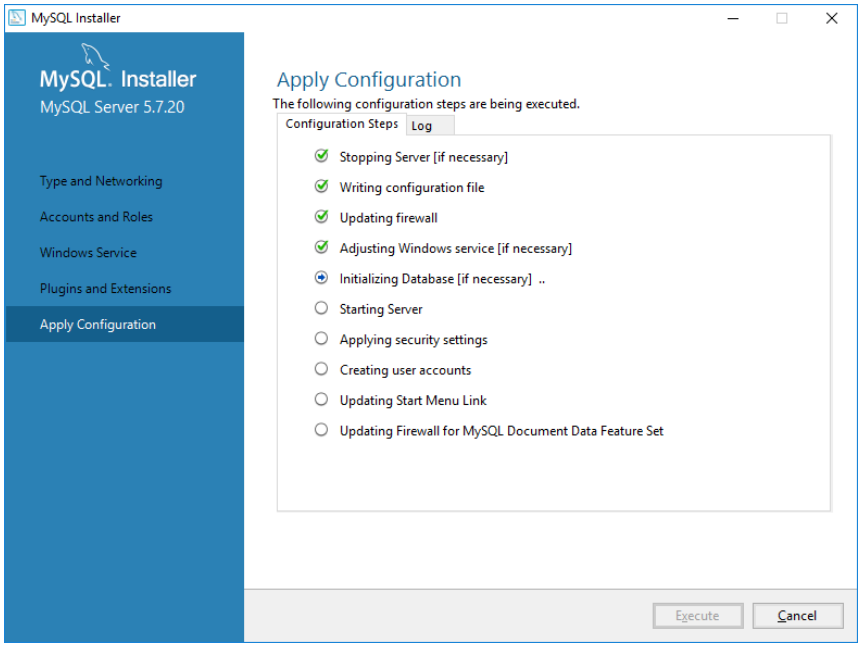


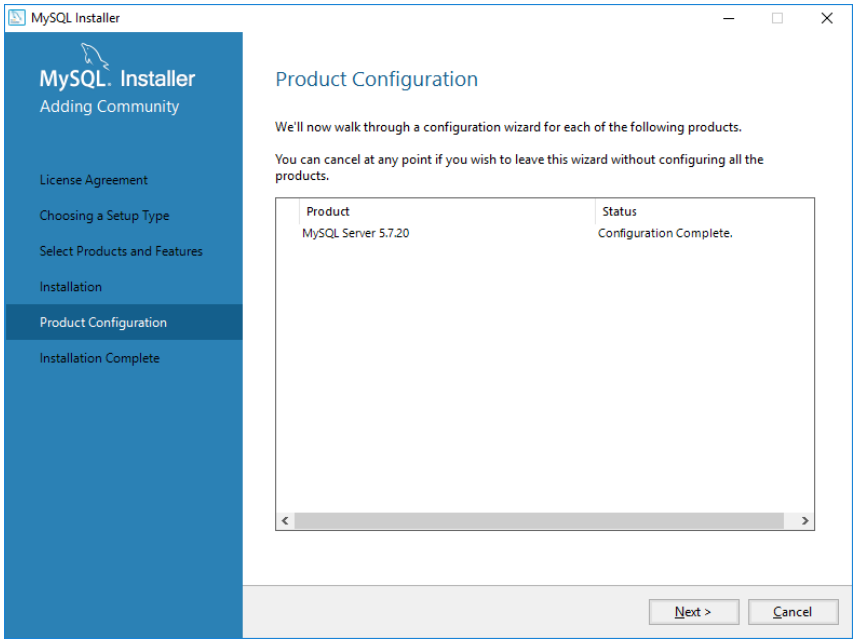

MySQL Installer
MySQL. Installer
MySQL Server 5.7.20
Type and Networking
Accounts and Roles
Windows Service
Plugins and Extensions
Apply Configuration
Windows Service
Configure MySQL Server as a Windows Service
Windows Service Details
Please specify a Windows Service name to be used for this MySQL Server instance. A unique name is required for each instance.
Windows Service Name: MySQL57
Start the MySQL Server at System Startup
Run Windows Service as ...
The MySQL Server needs to run under a given user account. Based on the security requirements of your system you need to pick one of the options below.
Standard System Account
Recommended for most scenarios.
Custom User
An existing user account can be selected for advanced scenarios.
< Back
Next >
Cancel

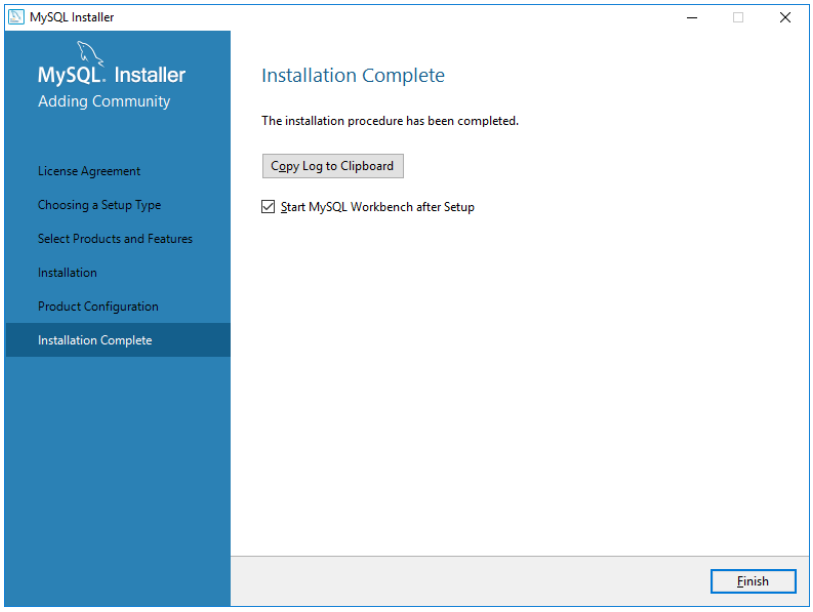

MySQL Installer
MySQL. Installer
MySQL Server 5.7.20
Type and Networking
Accounts and Roles
Windows Service
Plugins and Extensions
Apply Configuration
Plugins and Extensions
MySQL as a Document Store
Use the following controls to select how you would like to connect to this server.
Enable X Protocol / MySQL as a Document Store
Port Number: 33060
Open Firewall port for network access
Starting with MySQL Server 5.7, MySQL supports document store development. In order to provide a complete document store/NoSQL experience there is a new communications protocol called the X Protocol. The expanded capabilities of the X Protocol enable us to provide modern developer APIs with features such as asynchronous calls, pipelining, and more. In addition to implementing document collections, the new X DevAPI also supports relational and combined document store/relational capabilities. Now developers, designers and DBAs can deploy MySQL databases that implement document store, relational, or hybrid document/relation models.
Click here to view MySQL as a Document Store online documentation
< Back
Next >
Cancel

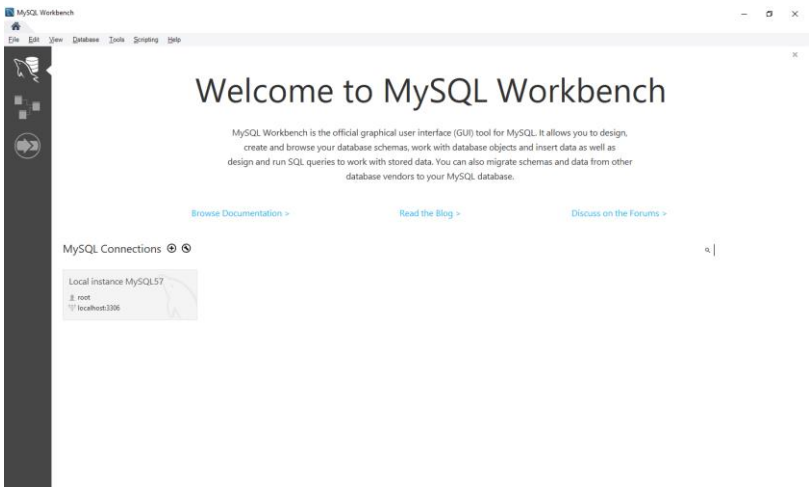


Click pe butonul “Execute”

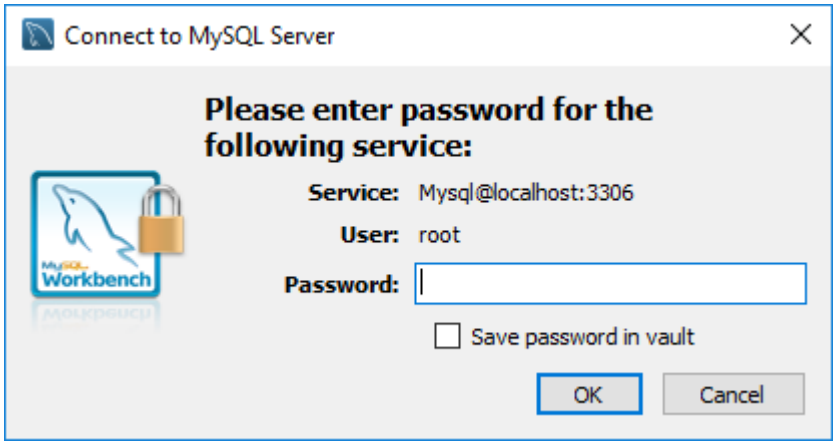


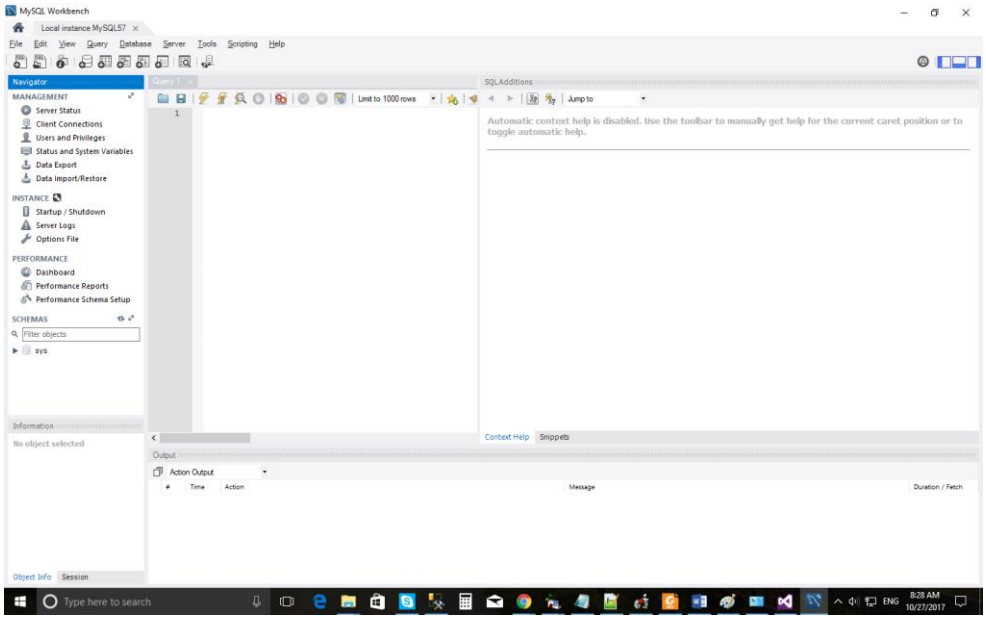






Click pe “Local instance MySQL > root”. Vi se va cere parola:


Connect to MySQL Server
Please enter password for the following service:
Service: Mysql@localhost:3306
User: root
Password:
Save password in vault
OK
Cancel


MySQL Workbench
Local instance MySQL57
File Edit View Query Database Server Tools Scripting Help
Navigator
MANAGEMENT
Server Status
Client Connections
Users and Privileges
Status and System Variables
Data Export
Data Import/Restore
INSTANCE
Startup / Shutdown
Server Logs
Options File
PERFORMANCE
Dashboard
Performance Reports
Performance Schema Setup
SCHEMAS
Filter objects
sys
Information
No object selected
Object Info Session
Query 1
Limit to 1000 rows
SQLAdditions
Jump to
Automatic context help is disabled. Use the toolbar to manually get help for the current caret position or to toggle automatic help.
Context Help Snippets
Output
Action Output
Time Action Message Duration / Fetch
Type here to search
ENG
8:28 AM
10/27/2017