Baze de Date

# Curs

Profesor: Iulia Banu (email: [th\_iulia84@yahoo.com](mailto:th_iulia84@yahoo.com))

Slide-uri: <https://tinyurl.com/cursuribd>

Pentru partea de teorie recomand acest curs de pe Udacity: [Database Systems Concepts & Design](https://www.udacity.com/course/database-systems-concepts-design--ud150)

Vom învăța:

* proiectarea bazelor de date
* baze de date relaționale
* **S**tructured **Q**uery **L**anguage (**SQL**).

Implementare de software de baze de date: vom folosi [Oracle Database](https://www.oracle.com/database/).

## Curs 1

### Sistem de Gestiune a Bazei de Date (SGBD) / Database Management System (DBMS)

Un sistem de **definire** și **manipulare** a **datelor**.

Operații care se pot efectua:

* **Stocarea** datelor
* **Accesarea** datelor
* **Actualizarea** datelor

Proprietăți care trebuie menținute:

* Evităm **redundanța** (nu stocăm în mai multe locuri aceeași informație)
* Evităm **inconsistențele** (nu stocăm informații contradictorii)

Permite accesul la date a mai mulți utilizatori în același timp (**concurență**).

Oferă **securitate** și **backup + recuperarea** datelor.

Oferă limbaje declarative pentru definirea, accesarea, manipularea și controlarea datelor:

* DDL (Data Definition Language)
* DQL (Data Query Language)
* DML (Data Manipulation Language)
* DCL (Data Control Language)

Conține **metadate** (informații despre baza de date, cum arată, ce tipuri de date sunt folosite, etc.)

### Istoria DBMS și exemple

* **Flat files**
  + fișiere .csv (comma separated values), în care păstrăm pe fiecare linie o valoare
* **Hierarchical systems**
  + sistemul de fișiere de pe un calculator
  + XML
* **Object oriented**
  + **Realm** database pe Android/iOS, clasele și obiectele sunt elementele bazei de date
* **Relational DBMS**
  + cele bazate pe SQL
  + bazate pe tranzacții
  + oferă **ACID**:
    - **Atomicity**
    - **Consistency**
    - **Isolation**
    - **Durability**
* **NoSQL (Not Only SQL)**
  + structură mai flexibilă
  + mai ușor de scalat
  + oferă suport pentru **sharding**, prin care baza de date este împărțită pe mai multe servere
  + oferă **BASE**:
    - **Basically Available**
    - **Soft state**
    - **Eventually consistent**
* **Hybrid**
  + combinație între SQL și NoSQL
  + combină și on-disk database cu in-memory database

### Diagrama Entitate-Relație (ER)

Reprezentare vizuală a modelului bazat pe **entități** și **relații** între entități.

Descrie structura logică a unei baze de date relaționale.

Procesul de dezvoltare a bazei de date este următorul:

* Se culeg **user story** și **software requirements**
* Se realizează **diagrama entitate-relație**
* Se definește **schema** bazei de date relaționale

O diagramă este formată din:

* **entități:** obiecte, persoane, evenimente, concepte din lumea reală; de obicei sunt substantive
* **relații:** leagă entitățile între ele; de obicei sunt verbe
* **atribute:** descriu entitățile sau relațiile

### Entități

Reprezentate ca dreptunghiuri pe diagramă.

Într-o BD relațională corespund unui **tabel**. Fiecare instanță corespunde unei **linii** din tabel.

Pot avea o **cheie primară**, adică un atribut sau grup de atribute care identifică în mod unic o instanță a entității.

Există entități tari (strong) și entități slabe (weak). Cele tari sunt de sine stătătoare, cele slabe depind de existența altor entități.

De exemplu, un avion poate fi o entitate tare, dar un loc din acel avion este o entitate slabă.

### Cheie primară

Identificator unic al unui obiect, trebuie să fie știut tot timpul.

Ar trebui să fie simplă, neambiguă, și imutabilă.

### Cheie străină

Identifică unic un obiect din alt tabel.

### Coloane autogenerate

În funcție de SGBD-ul ales, avem acces la câteva coloane care întotdeauna există, și au semnificații speciale.

În MySQL avem acces la coloanele

* DB\_ROW\_ID: număr care crește monoton pe măsură ce se inserează rânduri
* DB\_ROLL\_PTR: pointer la *roll record*
* DB\_TRX\_ID: identificator unic pentru ultima tranzacție care a inserat sau actualizat un rând

În Oracle avem:

* rowid: identificator unic de tip string care conține informații despre *block*, *row* și *file*
* rownum: numărul rândului în tabel

În Mongo avem:

* ObjectID

### Relații

Reprezentate ca arce orientate pe diagramă.

Într-o BD relațională pot fi reprezentate printr-o coloană de tip **cheie străină**, sau ca un tabel.

Poate avea o anumită cardinalitate:

* limita inferioară este obligatorie (must have at least N)
* limita superioară este opțională (may have up to N)

### Indici

Structuri de date specifice care ajută la căutarea bazată pe o cheie.

Indicii pot îmbunătăți performanța, dar nu afectează comportamentul vizibil al unui tabel. Cu alte cuvinte, pot fi creați și șterși fără a afecta utilizatorii tabelului.

Avantaje:

* optimizează căutările
* optimizează join-urile
* optimizează sortarea/gruparea

Dezavantaje:

* pot încetini comenzile DML
* consumă mai multă memorie

E bine să fie folosite dacă estimăm că rezultatele unui query returnează de obicei mai puțin de ~15% din datele din tabel (deci se merită să evităm căutarea prin tot tabelul).

### Tipuri de indici

* **Clustered index:** definește ordinea în care sunt stocate datele pe disc. Se creează automat când fixăm o cheie primară.
* **B-Tree Index:** un arbore echilibrat de căutare. Frunzele sunt pointeri la rândurile respective.
* **Reverse Index:** un arbore echilibrat de căutare cu elemente în ordine inversă.
* **Bitmap Index:** se folosește când o coloană are un număr limitat de valori distincte (enumerație de câteva valori)

## Curs 2 & 3

### Modelul relațional (bazat pe algebra relațională)

O bază de date este o colecție de relații (! nu sunt aceleași cu cele din diagrama ER).

**Schema** relației: numele unei relații, și atributele acesteia (i.e. tabelul și câmpurile lui)

**Domeniul** unui atribut: mulțimea de valori pe care le poate avea un atribut.

### Constrângeri de integritate

* Constrângeri de **domeniu**
  + coloana poate fi setată să aibă valori distincte, sau să nu conțină null, etc.
  + fiecare atribut trebuie să aibă un **unic tip de date** (nu pot să am coloană și cu numere și cu stringuri)
  + trebuie să respecte tipul de date al atributului
  + la numere, trebuie să aibă precizia/numărul de cifre specificat
  + la stringuri, trebuie să respecte lungimea maximă
* Constrângeri de cheie
  + **cheie primară** implică **unică** și **nu e null**
* Constrângeri de **integritate referențială**
  + o **cheie străină** fie este null, fie corespunde unei cheie primare a unui obiect existent din celălalt tabel

### Schema relațională

Fie D\_1, D\_2, …, D\_n domeniile de valori ale atributelor.

Fie R o relație pe D\_1 x D\_2 x … x D\_n.

Fie A\_1, …, A\_n valori ale atributelor.

Notăm R(A\_1, A\_2, …, A\_n) faptul că A\_1, …, A\_n se află în relația R.

### Operatori

Pentru a obține datele de care avem nevoie, compunem unii dintre următorii operatori:

* Operatori pe algebra mulțimilor
  + Reuniune: UNION pe mulțimi
  + Intersecție
  + Produs cartezian: full outer join
  + Diferență: MINUS pe mulțimi
* Proiecție: când alegem anumite coloane ale unui tabel
* Selecție: când folosim WHERE pentru a filtra anumite rânduri din tabel
* Join: când facem un inner sau left/right outer join
* Împărțire

### Transformarea unei diagrame ER într-un model relațional

#### Transformarea entităților

Entitățile tari se transformă în tabele independente. Cheile lor primare nu conțin chei străine.

Entitățile slabe se transformă în tabele care depind de alte tabele. Cheile lor primare fac referire la cheia entității tari asociate.

Sub-entități: se transformă într-unul sau mai multe tabele, atribute. Cheile lor primare pot fi și o cheie străină.

#### Transformarea relațiilor

* Relații **1-to-1** și **1-to-many**: se transformă în **chei străine**
* Relații **many-to-many** sau relații **ternare**: se transformă în tabele asociative, conțin chei străine la tabelele pe care le leagă

#### Transformarea atributelor

* Atribute simple: se transformă într-o coloană.
* Atribute cu mai multe valori (cu mai multe tipuri de date): se transformă în mai multe coloane sau într-o entitate slabă, reprezentată printr-un tabel.

## Curs 4

### Tranzacții

O tranzacție este o mulțime de operații aplicate bazei de date, formată din mai multe comenzi de **inserare**, **actualizare** și **ștergere**.

Sunt delimitate de comenzi de tipul **begin transaction** / **end transaction**.

Fie toate operațiile se termină cu succes, fie nicio modificare nu este salvată în baza de date.

Un sistem tranzacțional trebuie să asigure

* abilitatea de a efectua tranzacții concurent
* consistența datelor în caz de eșec

### Proprietățile tranzacțiilor (ACID)

* **A**tomicity
  + Se aplică toate operațiile, sau niciuna (sunt tratate ca o singură unitate indivizibilă).
  + Nu contează dacă eroarea este produsă de programator, sau externă.
* **C**onsistency
  + Dacă tranzacția se aplică pe o bază de date consistentă, la finalul tranzacției trebuie să fie tot consistentă.
  + Toate constrângerile de tip PK/FK, unique, not null, check, etc. se respectă.
  + Baza de date poate fi pe parcursul tranzacției într-o stare inconsistentă, dar asta nu este vizibil din exterior
  + Valorile de dinaintea tranzacției se păstrează într-un log file, pentru a permite recuperarea datelor
* **I**solation
  + Sistemul trebuie să se asigure că tranzacțiile nu pot interfera una cu cealaltă.
  + Pentru orice două tranzacții T\_i și T\_j, fie T\_i se execută complet înainte lui T\_j, fie invers, dar nu se pot executa ambele în același timp.
* **D**urability
  + După ce se termină o tranzacție, schimbările rămân salvate, chiar dacă pică sistemul
  + Informațiile pot fi recuperate

### Stările tranzacționale

* La început sistemul se află în starea *Active*
* După ce s-au executat calculele din tranzacție, sistemul se află în starea *Partially committed*
* Dacă nu apar alte erori, se aplică modificările și trecem în starea *Committed*
* Dacă apare vreo problemă cu tranzacția, trecem în modul *Failed*
* Sistemul dă undo la modificări și ajungem în starea *Aborted*

### Tranzacții concurente

Pentru a reduce timpul de răspuns și pentru a îmbunătăți performanța, sistemul poate încerca să ruleze tranzacții în același timp.

Dacă se încalcă principiul **Isolation**, există riscul ca baza de date să ajungă într-o stare inconsistentă.

Pentru asta, sistemul de baze de date împarte tranzacția în **instrucțiuni** și încearcă să le ordoneze pentru a păstra ordinea cronologică a schimbărilor. Această ordonare se numește **Schedule**.

### Schedule serializabil

Spunem că un *schedule* este serializabil dacă este echivalent cu o secvență neconcurentă de instrucțiuni.

Există diferite tipuri de echivalență:

* Serializabilitate de conflict
  + Dacă interschimbăm operațiile care nu intră în conflict, obținem un nou schedule conflict-echivalent
  + Sistemul poate să reprezinte tranzacțiile ca un graf orientat, unde un arc înseamnă că o tranzacție intră în conflict cu altă tranzacție, și să facă o sortare topologică pentru a găsi ordinea optimă de execuție
* Serializabilitate de view
  + Ne bazăm că o tranzacție termină de scris niște valori de care ar avea nevoie o altă tranzacție.
  + A găsi soluția optimă din acest punct de vedere este o problemă NP-completă.

### Nivele de izolare

**Izolarea** se reduce la a executa o tranzacție **de parcă** nu ar fi alte tranzacții care rulează în același timp.

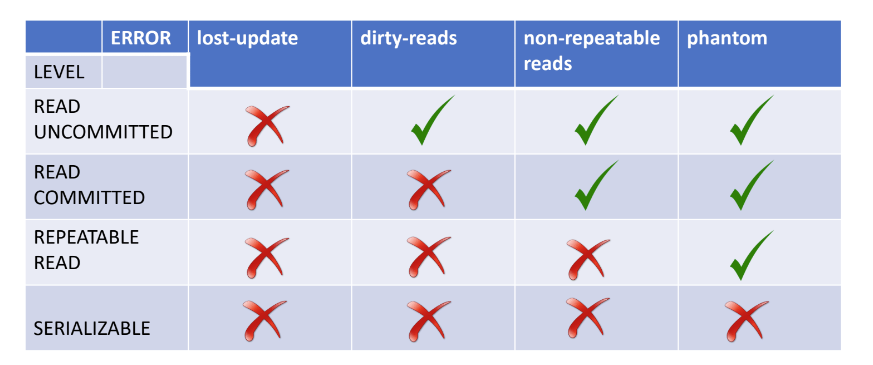
Nivelul de izolare al unei tranzacții determină cât de repede se termină și câte tranzacții putem executa în același timp.

* Dacă executăm o singură tranzacție pe sistem, aceasta are acces la toate resursele și se termină foarte repede, dar e numai una.
* Dacă executăm mai multe în același timp, fiecare durează mai mult.

Principalele tipuri de erori:

* lost update: facem o modificare, dar aceasta nu este salvată
* dirty read: citim date mai vechi, pentru că încă nu s-au propagat modificările în sistem
* non-repeatable read: citim o valoare temporară
* phantom: calculăm o valoare cu date care nu ar fi trebuit să existe încă (de exemplu, AVG > MAX pentru că au apărut linii între timp)

Principalele nivele de izolare:



### Implementarea izolării

* Versionare: fiecare stare a bazei de date primește o versiune distinctă. O tranzacție citește o anumită versiune și produce o nouă versiune.
* Locking
  + Un lock/mutex garantează că o singură tranzacție accesează o resursă la un moment dat
  + Un lock poate fi *shared* (mai mulți cititori în același timp) sau *exclusive* (un singur scriitor)
  + O tranzacție poate rămâne blocată în *deadlock*
* Snapshot isolation
  + Se face un „snapshot” a bazei de date la începutul tranzacției
  + Tranzacția lucrează pe acel snapshot
  + Tranzacția modifică direct datele din spațiul de lucru, până dă commit la final
  + Se implementează cu versionare bazată pe timestamp

## Curs 5

### Normalizare

Normalizarea înseamnă să organizezi informația dintr-o bază de date într-un mod care elimină redundanța și previne anomaliile produse de manipularea datelor.

De obicei constă în descompunerea unei relații (un tabel) într-un mod care nu pierde datele.

### Descompunere

Pentru a elimina o informație redundantă:

1. Aplicăm o **proiecție** și extragem una sau mai multe coloane într-un **tabel nou**
2. Printr-un JOIN putem obține rândurile inițiale

Dacă în urma JOIN-ului obținem fix relația inițială, spunem că descompunerea este fără pierderi (lossless).

Dacă obținem elemente în plus, care nu erau în relația inițială, spunem că descompunerea este cu pierderi (lossy).

### Dependențe funcționale

Dacă valoarea dintr-o coloană se poate calcula ca o funcție a celorlalte coloane, atunci aceasta depinde funcțional de celelalte.

### Forme Normale (Normal Forms)

Dacă sunt îndeplinite anumite condiții, putem spune că baza de date se află într-o anumită formă normală.

Pe măsură ce respectăm forme normale din ce în ce mai mari, crește integritatea și performanța bazei de date.

### Forma Normală 1 - Atribute atomice

Fiecare atribut trebuie să aibă o singură valoare (trebuie să fie atomic).

Fiecare atribut are doar valori atomice.

Fiecare atribut are doar valori din domeniul său.

O bază de date relațională întotdeauna este cel puțin în Formă Normală 1, pentru că în SQL nu poți avea o coloană care să aibă mai multe tipuri de date, și într-o coloană ai doar valori ale tipului de date respectiv.

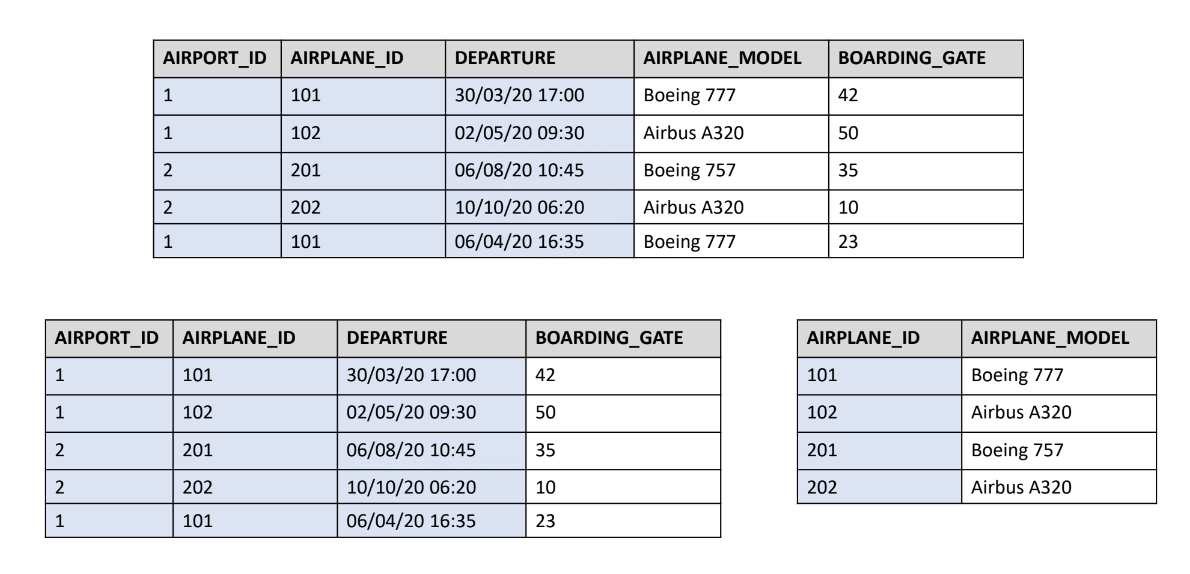
### Forma Normală 2 - Fără dependențe parțiale

FN 2 implică FN 1.

Trebuie să nu existe atribute non-cheie care să depindă de (unele dintre) atributele cheii.

Cu alte cuvinte, nu ar trebui că o coloană oarecare să fie doar o funcție a cheii.

Exemplu: modelul avionului depinde tot timpul de ID-ul avionului (pentru că un avion, identificat prin ID, nu poate avea mai multe modele). Soluția este să mutăm în alt tabel datele despre modelul fiecărui avion.



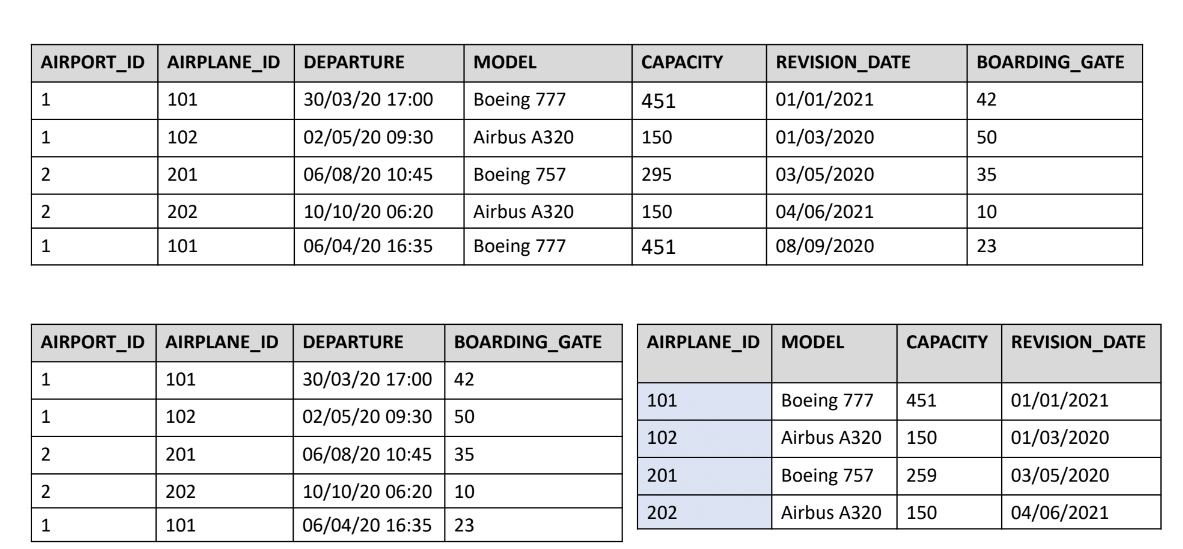
### Forma Normală 3 - Fără dependențe tranzitive

FN 3 implică FN 2.

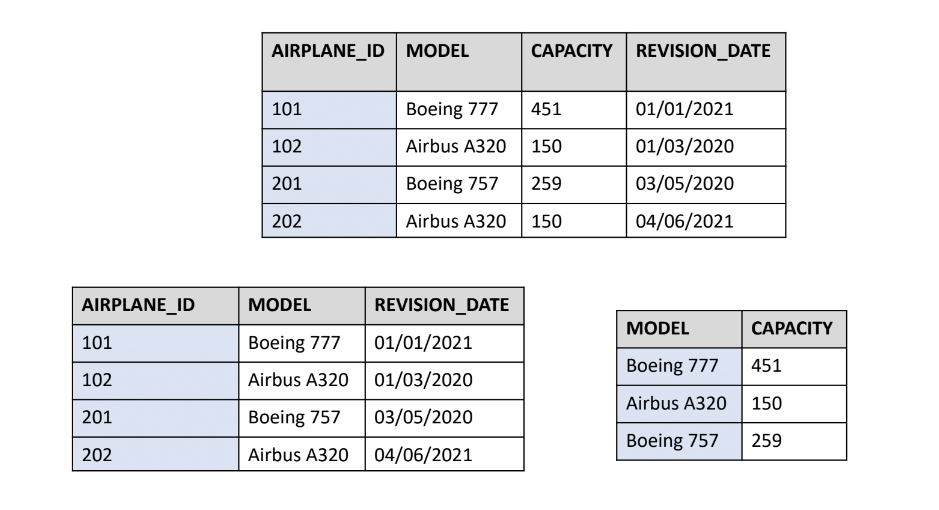
Nu există dependențe tranzitive.

În acest exemplu, capacitatea unui avion depinde de modelul acestuia, care depinde de ID-ul avionului.

Mai întâi separăm într-un nou tabel datele despre modelul de avion.



Apoi eliminăm dependența tranzitivă între capacitate și model, creând un nou tabel:



Acum baza de date este în FN3.

## Curs 6

### Executarea interogărilor

1. Sistemul primește ca input o **interogare în SQL**.
2. Aceasta este tradusă de un **parser** într-o **expresie** dealgebră relațională.
3. Această expresie este **optimizată**, și transformată într-un **plan de execuție**.
4. Planul de execuție este **evaluat**.
5. Rezultatul este returnat ca output.

### Proprietăți din algebra relațională

Putem să înlocuim expresia obținută din interogarea SQL cu una echivalentă, care este mai eficientă și accesează mai puține date.

Lista cu proprietățile se găsește în curs.

### Reguli generale de optimizare ale expresiilor

* Mai întâi se execută selecțiile (adică condițiile WHERE), pentru a reduce dimensiunea relației (numărul de rânduri posibile).
* E bine să se evite cross-join (produsul cartezian), de preferat sunt join-urile obișnuite
* Mai întâi se execută join-ul care ar produce o relație mai mică

### Optimizări la evaluare

Când trebuie evaluată expresia, algoritmul încearcă să mai aplice câteva optimizări:

* **Hash join**: dacă în join folosim o cheie, se alege tabelul mai mic și se construiește un hash table, prin care se optimizează join-ul.
* **Nested loops**: se încearcă combinarea buclelor imbricate.
* **Merge join**: se poate face un join bazat pe interclasare.

### Cum măsurăm costul unei cereri

Putem măsura:

* costul de Input/Output: cât accesăm disk-ul, câte blocuri se transferă, câte tupluri (rânduri) se citesc de pe disk
* timpul de execuție pe CPU: cât costă să procesăm un tuplu, cât costă să procesăm un element din index, cât durează să executăm o funcție

Statistici:

* per tabel
  + număr de rânduri
  + număr de blocuri
  + dimensiunea medie a unui rând
* per coloană
  + numărul de valori distincte
  + numărul de valori NULL
  + distribuția datelor
* per index
  + numărul de frunze
  + numărul de nivele
* per sistem

## Curs 7

### Bază de date relațională vs Big Data

|  |  |
| --- | --- |
| Bază de date relațională | Big Data |
| Date structurate | Date semi-structurate, heterogene, grafuri |
| Date dintr-o singură companie | Date din multe surse |
| Poate fi păstrată pe un singur server | Necesită mult paralelism pentru stocare și procesare |
| Scalare de obicei pe verticală (upgrade la hardware) | Sharding, sisteme de stocare bazate pe cheie-valoare, sisteme bazate pe documente |

### Map-reduce

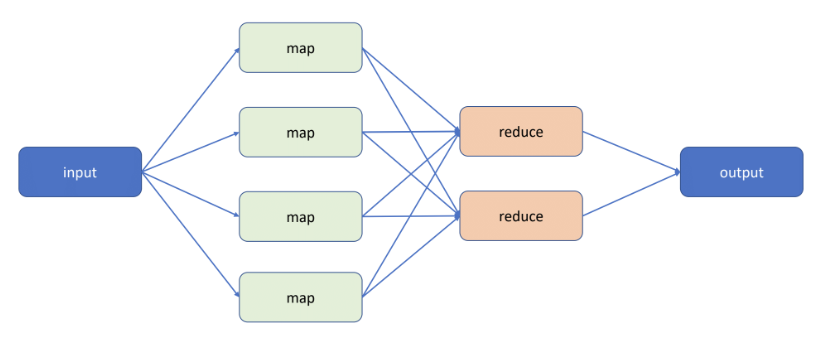
Este o paradigmă, apărută la Google, acum există și framework-uri open source (de exemplu Apache Hadoop).

Construiește algoritmi folosiți pentru procesarea în paralel, bazați pe programarea funcțională.

Input-ul poate veni în diferite forme. Această metodă poate procesa un volum foarte mare de date.

Se bazează pe două etape:

* Faza **map**: se aplică o funcție pe datele de intrare. Aceasta se poate aplica în paralel, pe mai multe noduri.
* Faza **reduce**: se aplică o funcție de agregare pe outputul de la etapa anterioară.



### Hadoop

Pentru a folosi această paradigmă se poate folosi [Hadoop](https://hadoop.apache.org/), care este scris în Java, cu interfețe de C++ și Python.

Hadoop este format din două componente:

* MapReduce
* Hadoop Distributed File System (HDFS)
  + Un sistem de fișiere distribuit pe mai multe mașini
  + Rezistent la erori

Arhitectură master-slave: un nod controlează celelalte noduri.

### Index inversat în MapReduce

Putem să luăm și un singur input, și să-l distribuim pe mai multe mașini care rulează map și reduce.

De exemplu, când căutăm pe Google, un singur query este distribuit pe mai multe servere care găsesc rezultate (map) și returnează o pagină cu rezultate (reduce).

### Operatori SQL implementați în MapReduce

Putem simula orice operație de SQL (selecție, proiecție, join, etc.) folosind map reduce.

De exemplu, în faza *map* extragem din baza de date cheia pe care vom face join și datele asociate.

În faza *reduce* putem face join pe cheile care se potrivesc.

## Curs 8

### NoSQL (Not Only SQL)

Schema este **flexibilă**, se adaptează datelor. E **ușor de migrat** (adică de actualizat baza de date la o nouă schema).

Datele pot fi semi-structurate, complexe, incluse una într-alta.

**Nu** se folosește **SQL** (cum era la relational database).

De obicei **nu** are suport pentru **tranzacții**. Unele constrângeri ACID au trebuit relaxate pentru a face sistemul **scalabil**.

Se pune mare accent pe **performanța mărită** oferită de NoSQL.

Multe implementări sunt **open source** sau oferă un **API** public.

### Bază de date cheie-valoare (Key-Value Database)

Oferă două operații în principal:

* stochează la o anumită cheie o valoare: **put(key, value)**
* returnează valoarea asociată cu o anumită cheie: **get(key)**

**Exemple:** BigTable, Apache HBase, Dynamo, Cassandra, MongoDB, Azure

### Baze de date orientate pe documente = Document stores

Datele sunt reprezentate într-un mod unificat, de exemplu în format JSON.

Se pot executa interogări simple pe datele stocate (de exemplu, căutare după valoarea unui câmp).

**Exemplu:** MongoDB

### Partitioning

Partiționarea înseamnă împărțirea în submulțimi disjuncte a datelor.

În cadrul bazelor de date cheie-valoare, înregistrările sunt împărțite pe mai multe mașini. Fiecare accesează și actualizează o parte din toate înregistrările.

Tipuri de partiționare:

* pe orizontală (de exemplu, **sharding**)
* pe verticală

### Sharding

Partiționarea se realizează pe atribute cunoscute ca **partitioning keys** sau **shard keys**.

Metode de sharding:

* partiționare pe intervale (**range** partitioning): datele sunt împărțite pe intervale ale cheii.
* partiționare după hash (**hash** partitioning): datele sunt distribuite uniform, dar dacă vrem să accesăm datele consecutive dintr-un interval, trebuie să accesăm mai multe noduri.

Probleme:

* trebuie folosită replicarea datelor în caz că un nod cedează
* această replicare trebuie să fie consistentă
* JOIN pe tabelele distribuite depinde de viteza comunicării pe rețea

### Sharding în MongoDB

Datele se împart în **chunks**, un interval bazat pe o cheie comună a documentelor.

Arhitectură:

* Mongos: primesc cererile și le dirijează
* Servere de configurare
* Shards

Dacă o interogare nu se aplică la un singur shard, atunci nodurile mongo vor face broadcast la toate shardurile.

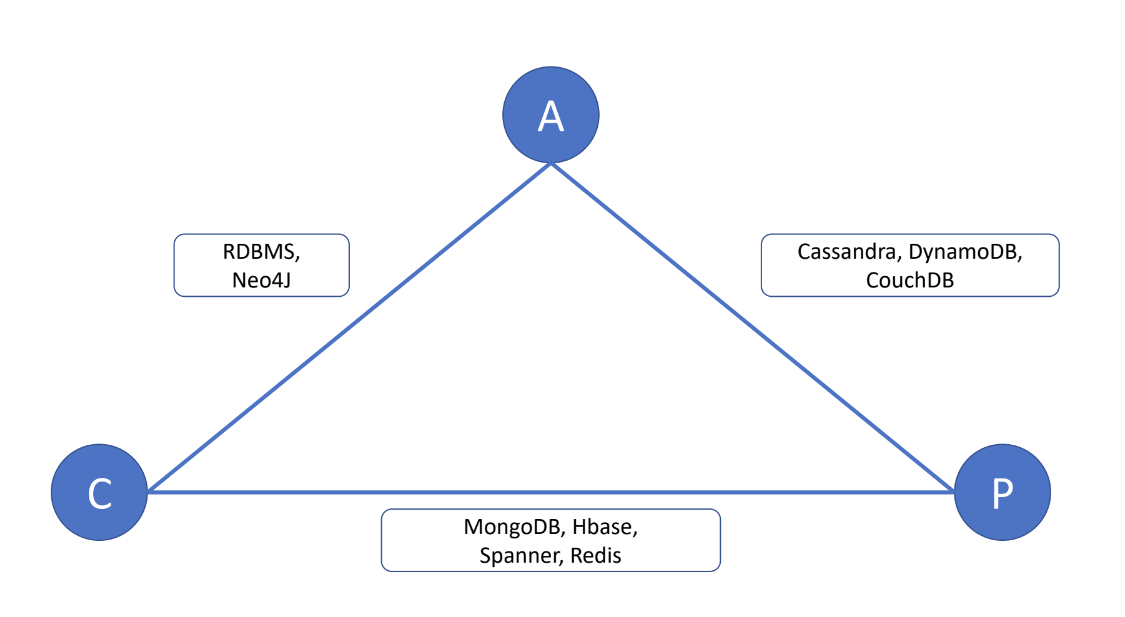
### Teorema CAP

O bază de date **distribuită** nu poate garanta mai mult de **două** dintre următoarele:

* **Consistency**: citirea returnează datele așa cum arată ele după cea mai recentă scriere sau eroare.
  + **Linearizable consistency**: odată ce o operație este completă, este vizibilă la toate nodurile.
  + **Eventual consistency**: dacă nu se mai fac alte actualizări, eventual efectele unei operații ajung să fie vizibile în toate nodurile.
* **Availability**: orice cerere primește un răspuns care nu este eroare.
  + **High availability**: nodurile care încă funcționează (non failing) returnează un răspuns la orice cerere.
* **Partition tolerance**: sistemul continuă să funcționeze în ciuda pierderii unui număr arbitrar de mesaje.

Cele 3 combinații posibile de câte două proprietăți de mai sus sunt:

* **CP**: sacrific availability pentru consistency și partition tolerance
* **AP**: sacrific consistency pentru availability și partition tolerance
* **CA**: sacrifici partition tolerance pentru consistency și availability



Extensie a teoremei CAP: [PACELC theorem](https://en.wikipedia.org/wiki/PACELC_theorem) (ia în considerare și compromisul între latency și consistency în cazul nepartitiționat)

### CAP în MongoDB

Mongo este o bază de date care a optat pentru C+P.

Pentru fiecare cluster avem un **nod primar** și mai multe **noduri secundare**. Nodul primar primește tranzacțiile de scriere și le efectuează. După ce termină, nodurile secundare îl copiază.

Dacă pică nodul primar, nodul secundar care are cea mai recentă modificare a datelor îi ia locul.

Availability se pierde deoarece clusterul este disponibil doar după ce toate nodurile secundare au replicat nodul primar.

### CAP în Cassandra

Cassandra a optat tot pentru C+P.

Este un sistem **eventually consistent**: nu e garantat că toate nodurile replicate au aceleași date.

Nivelul de consistență: numărul de noduri replicate care sunt necesare pentru a realiza o operație read/write

* ONE: se folosește cel mai apropiat nod
* QUORUM: se sincronizează mai multe noduri între ele și câștigă majoritatea

### Nivele de consistență

* **Strictă**: se respectă un ceas global, toate scrierile sunt imediat vizibile de toate procesoarele.
* **Secvențială**: ordine globală între operațiile de scriere.
* **Atomică** (linearizabilitate): ordine globală între operațiile care nu se suprapun în timp.
* **Cauzală**: ordine globală între operațiile de scriere care se suprapun.
* **Eventual consistent**: dacă nu se efectuează nicio scriere pentru o perioadă de timp, eventual toate nodurile vor ajunge să vadă ultima valoare scrisă.

### Basically Available, Soft state, Eventually consistent (BASE)

La fel cum bazele de date SQL garantau proprietățile ACID, bazele de date NoSQL oferă BASE.

* **Basically Available**: latență scăzută, disponibilitate mare
* **Soft state**: schimbările se propagă printre noduri automat
* **Eventually consistent**

### Privire de ansamblu MongoDB

Noțiunile din MongoDB [se potrivesc](https://docs.mongodb.com/manual/reference/sql-comparison/) cu noțiunile unei baze de date relaționale:

* **Document**: mai multe perechi cheie-valoare, asemănătoare cu un obiect JSON ⇔ rând într-un tabel
* **Colecție**: mai multe documente, documentele pot avea câmpuri diferite ⇔ tabele
* **Câmp** într-un document JSON ⇔ coloane
* **$lookup** și documente **embedded** ⇔ JOIN-uri

API: <https://docs.mongodb.com/manual/core/databases-and-collections/>

Fiecare obiect inserat primește automat o cheie **\_id**, creată pe baza:

* timpului inserării (4 bytes)
* un număr aleator (5 bytes)
* un contor care crește automat de la o valoare aleatoare (3 bytes)

Tipuri de index:

* **single field** index
* **compound** index
* **multi key** index
* **geospatial** index
* **text** index
* **hashed** index

## Curs 9

### Filtru Bloom

Structură de date probabilistă, folosită pentru a verifica apartenența la o mulțime.

Pentru o mulțime *S* cu *n* valori, se alochează un număr binar cu *constantă \* n* biți.

Pentru fiecare valoare din *S* se calculează un *hash*, care reprezintă un bit. Bitul corespunzător acelui hash se setează ca 1.

Pentru a verifica dacă o valoare aparține mulțimii, se calculează hash-ul și apoi se verifică dacă bitul respectiv e 1.

Există riscul să apară **false positives**: o valoare care nu ar aparține mulțimii poate avea același hash cu una care aparține.

Îmbunătățire: putem folosi mai multe funcții de hashing independente.

Pentru fiecare valoare, calculăm *hash1*, *hash2*, *hash3*, … etc. și setăm biții respectivi să fie 1.

Când vine o nouă valoare, trebuie ca biții calculați pentru toate funcțiile de hash să fie 1.

Se pot doar adăuga elemente (odată ce a fost adăugat la filtru nu poate fi eliminat).

Poate fi folosit doar pentru a testa dacă un element aparține mulțimii.

Mai mult spațiu => crește **consumul de memorie** dar crește și **acuratețea**.

Mai multe funcții de hash => crește **latența** (durează mai mult să se calculeze) dar crește și **acuratețea**.

Formule pentru independența funcțiilor de hash, false positive rate, acuratețe - în curs.

### Log Structured Merge-tree

Folosită pentru a optimiza accesarea datelor pe disk.

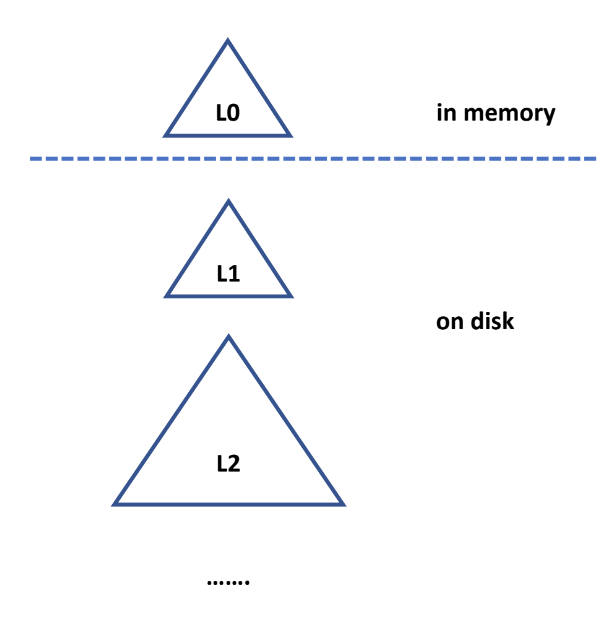
Implementări: în BigTable, LevelDB, Apache Cassandra, etc.

#### Implementare

Folosește **arbori B+** (de la binary+): nodurile pot avea mai mult de 2 copii.

Primele câteva nivele ale arborelui sunt reținute în memorie.

Celelalte nivele se află pe disc. Nodurile de pe același nivel sunt aflate unul după altul pe disk (pentru a crește viteza de acces).



### Materialized view

Feature al implementărilor SQL, prin care se salvează într-un nou tabel rezultatele unui view.

Conțin date **redundante** (pentru că am putea oricând să le recalculăm din tabelele sursă, folosind definiția lor).

Se pot actualiza:

* imediat (**immediate** view refresh)
* întârziat (**deferred** view refresh)
* incremental: se modifică doar părțile afectate

Se pot folosi pentru: JOIN, selecție, proiecție, agregare.

# Laborator

Laborant: Riclea A.

Laboratoare: <http://193.226.51.37/down/sql/index.html>

## De unde să învăț?

SQL se învață cel mai bine lucrând, câteva resurse ar fi:

* [Learn Intro to SQL Tutorials](https://www.kaggle.com/learn/intro-to-sql) și [Learn Advanced SQL Tutorials](https://www.kaggle.com/learn/advanced-sql)  
  (te învață să rulezi SQL pe Google BigQuery, și primești și niște diplome când le termini)
* rezolvând exercițiile din laboratoare
* Cursul de pe Udacity [SQL for Data Analysis](https://www.udacity.com/course/sql-for-data-analysis--ud198) (dați pe Start Free Course)
* Exerciții [de SQL de pe HackerRank](https://www.hackerrank.com/domains/sql)
* [PostgreSQL Exercises](https://pgexercises.com/)

**! Minim 6 prezențe pentru a promova**

**IP bază de date:** [**193.226.51.37**](http://193.226.51.37/)

! Lucrăm în [SQL Developer](https://www.oracle.com/ro/database/technologies/appdev/sql-developer.html)

**Exerciții rezolvate:** <https://github.com/GabrielMajeri/teme-fmi/tree/master/bd>