

## Baze de date - Introducere

### Definiții

O **bază de date** reprezintă o colecție de date inter-relaționate.

Un **SGBD** (Sistem de gestiune a bazelor de date) / **DBMS** (Database Management System) reprezintă un set de programe pentru organizarea, accesarea și manipularea bazelor de date.

Prin conceptul de **dată**, se înțelege înregistrarea într-un cod convenit a unei observații, obiect, fenomen, imagini, sunet, video, text etc. Datele reprezintă înregistrări brute. Prin acest lucru se înțelege faptul că acestea nu au fost încă procesate pentru a le evidenția semnificația.

Datele brute nu sunt de obicei folosite în această stare deoarece nu oferă o privire de ansamblu asupra a ceea ce reprezintă. Pentru a evidenția semnificația acestora, acestea trebuie prelucrate.

**Informația** este rezultatul prelucrării datelor brute cu scopul de a le evidenția semnificația.

*Datele reprezintă fundația informațiilor, iar informația reprezintă fundația cunoașterii.*

În funcție de informațiile dorite, procesarea datelor poate fi simplă (exemplu: o simplă organizare și agregare a acestora) sau extrem de complexă (exemplu: predicții și deducții bazate pe modelări statistice).

Informațiile pot fi folosite ca bază în procesele decizionale.

**Cunoașterea** furnizează informații și fapte despre un anumit subiect, într-un anumit context. Cunoașterea implică familiaritate, conștientizare și înțelegere a informațiilor. O caracteristică principală a cunoașterii este faptul că noi cunoștințe pot fi derivate din cunoștințe vechi.

### Exemplu:

Fig. 1 ilustrează o tabelă cu date despre cadrele didactice din cadrul unui departament. Această tabelă conține *date brute* despre cadrele didactice. Să presupunem că dorim să extragem *informația* referitoare la distribuția funcțiilor din cadrul departamentului. Pentru aceasta, datele brute trebuie mai întâi prelucrate. Informația extrasă ca rezultat al acestei prelucrări poate fi prezentată sub mai multe forme. Posibile reprezentări pot fi date în formă tabelară, ca în Fig. 2, sau în formă grafică, ca în Fig. 3 și Fig. 4.

ID	Nume	Prenume	Funcție
1	Popescu	Ion	asistent
2	Melinte	Doina	șef lucrări
3	Tabără	Andrei	profesor
4	Dorin	Mihai	șef lucrări
5	Luca	Marius	asistent
6	Oprescu	Viorel	asistent
7	Lupu	Silvia	conferențiar
8	Stan	Alexandru	șef lucrări
9	Truică	Violeta	șef lucrări
10	Severin	Dumitru	conferențiar

Fig. 1. Tabelă cadre didactice

Funcție	Contor	Procent
asistent	3	30%
șef lucrări	4	40%
conferențiar	2	20%
profesor	1	10%

Fig. 2. Distribuția funcțiilor (formă tabelară)

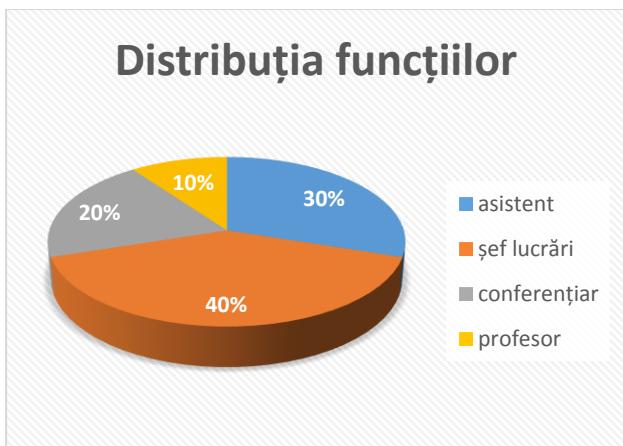


Fig. 3. Distribuția funcțiilor (pie chart)

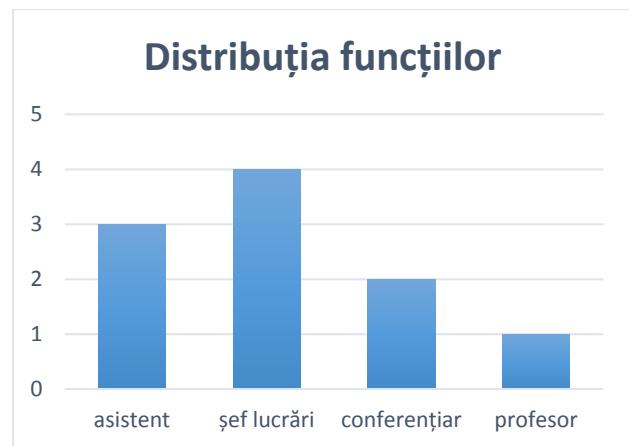


Fig. 4. Distribuția funcțiilor (column chart)

### Caracteristici și utilizări

SGBD-urile sunt concepute pentru a manipula cantități mari de date. Acest lucru presupune cel puțin:

- definirea structurilor de date pentru stocarea datelor
- furnizarea mecanismelor pentru manipularea acestora

În general, un SGBD oferă facilități suplimentare, cum ar fi:

- recuperarea datelor în cazul erorilor de sistem
- securitate pentru controlul accesului la date
- controlul accesului concurent la date pentru prevenirea anomaliei
- replicarea datelor etc.

Aplicații ale bazelor de date:

- bancare: tranzacții, clienți, depozite, carduri, acțiuni pe bursă, împrumuturi etc.
- companii aeriene: rezervări, orar, bilete etc.
- universități: înregistrarea studenților, cursuri, profesori, note etc.
- contabilitate: registre, facturi, balanțe, active, pasive etc.
- afaceri: clienți, produse, cumpărături etc.
- vânzări online: urmărirea comenzi, înregistrare clienți, coșuri de cumpărături, review-uri etc.
- producție: inventar, comenzi, furnizori, termene etc.
- resurse umane: angajați, bonusuri, salarii, taxe etc.
- telecomunicații: înregistrările log-urilor, carduri preplatite, informații despre rețea, trafic etc.
- etc.

## Necesitate

Înaintea apariției SGBD-urilor, stocarea și manipularea datelor se făcea folosind programe particularizate care lucrau direct cu fișiere. Apariția unor noi cerințe cât și necesitatea integrării unor noi tipuri de date presupunea crearea de noi programe specializate și fișiere aferente.

### Studiu de caz - facultate:

- adăugare studenți, profesori, cursuri
- înscrierea studenților la cursuri și generarea listelor de participanți
- adăugare note, calcul medii, generare rapoarte

Abordarea manipulării directe a fișierelor prezintă o serie de dezavantaje:

- date redundante și incoerente – formate diferite ale fișierelor ce conțin informații duplicate
- dificultate în accesarea datelor – un nou program trebuie scris pentru fiecare tip de interogare
- izolarea datelor – dificultate în accesarea datelor din fișiere codate în formate diferite
- probleme de integritate – constrângerile de integritate hard coded; dificil de adăugat altele noi
- probleme de atomicitate – dificil de implementat
- anomalii la accesul concurrent – dificil de implementat deoarece programe multiple pot accesa aceleași date
- probleme de securitate – accesul parțial asupra datelor este dificil de implementat
- etc.

SGBD-urile au apărut ca o necesitate pentru a adresa aceste dezavantaje.

## Niveluri de abstractizare

Una dintre caracteristicile principale ale unui SGBD este de a oferi utilizatorilor o vedere abstractă asupra datelor. Acest lucru presupune ascunderea detaliilor despre cum anume sunt stocate și manipulate datele.

### Nivelul fizic

- reprezintă cel mai jos nivel de abstractizare
- descrie cum sunt stocate datele
- conține structuri complexe de date

### Nivelul logic

- descrie ce date sunt stocate în baza de date și care sunt relațiile dintre acestea
- descrie baza de date folosind un număr mic de structuri simple
- ascunde complexitatea nivelului fizic oferind o independență fizică a datelor

### Nivelul view

- reprezintă cel mai înalt nivel de abstractizare
- descrie doar o parte din baza de date
- deși folosește structuri simple, nivelul logic poate avea o complexitate crescută în marile baze de date, datorită varietății tipurilor de date
- sistemul poate prezenta mai multe view-uri pentru aceeași bază de date
- poate fi folosit ca mecanism de securitate pentru prevenirea accesului utilizatorilor asupra unor părți din baza de date

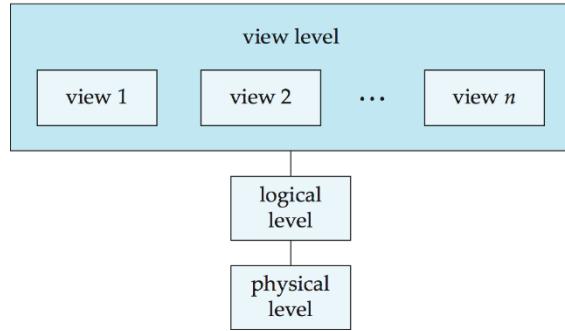


Fig. 5. Nivelurile de abstractizare a datelor [1]

#### *Paralelă cu limbajele de programare*

```

class Student implements IStudentBase, IStudentFull { // nivel logic
    int id;
    String nume;
    int an;
    String adresa;
    String email;
    String telefon;
    ...
}
interface IStudentBase { // nivel view
    int getId();
    String getNume();
    int getAn();
}
interface IStudentFull { // nivel view
    int getId();
    String getNume();
    int getAn();
    String getAdresa();
    String getEmail();
    String getTelefon();
}

Student s = getSomeStudent();
IStudentBase sBase = (IStudentBase)s; // view 1
IStudentFull sFull = (IStudentFull)s; // view 2

```

#### Nivelul fizic

Descrie detaliile de stocare a proprietăților obiectelor în locațiile de memorie.

#### Nivelul logic

Compilatorul ascunde de programator modalitatea de stocare a proprietăților obiectelor în memorie.

## Nivelul view

Programatorii pot oferi view-uri diferite asupra acelorași date prin intermediul interfețelor. În cazul aplicațiilor enterprise, acestea oferă interfețe remote diferite pentru accesul clienților în funcție de drepturile de acces ale acestora.

## Instante și scheme

Bazele de date se schimbă pe măsură ce noi informații sunt adăugate sau șterse din sistem. Colecția de date stocată într-o bază de date la un moment dat poartă denumirea de **instantă a bazei de date**. Structura per ansamblu a bazei de date poartă denumirea de **schema bazei de date**. În general, schemele bazelor de date nu sunt modificate în mod frecvent.

### *Paralelă cu limbajele de programare:*

Schema bazei de date corespunde declarațiilor variabilelor și tipurilor de date (clase, structuri etc.) dintr-un program.

Instanța bazei de date la un anumit moment corespunde valorilor tuturor variabilelor (inclusiv a variabilelor membre din clase, structuri etc.) la acel moment.

SGBD-urile au mai multe scheme corespunzătoare nivelurilor de abstractizare:

- schema fizică – descrie structura bazei de date la nivel fizic
- schema logică – descrie structura bazei de date la nivel logic
- scheme view – denumite și subscheme, descriu view-urile bazei de date

Dintre acestea, schema logică este cea mai importantă deoarece programatorii dezvoltă aplicații folosind această schemă. Schema fizică este ascunsă sub schema logică și poate fi schimbată fără a afecta aplicațiile deja dezvoltate. În acest context, se spune că aplicațiile prezintă **independență fizică a datelor**.

## Modele de date

**Modelul de date** este o colecție de instrumente conceptuale pentru descrierea datelor, a relațiilor dintre ele, a semanticii datelor și a constrângerilor de integritate. Acesta specifică o modalitate de a descrie structura bazei de date la nivelurile fizic, logic și view. Teoretic, fiecare nivel poate avea propriul model de date. În general, nivelul logic și nivelul view utilizează același model de date. Modelul de date de la nivelul fizic determină modalitatea în care datele sunt stocate, organizate, accesate și manipulate.

Definirea unui model de date presupune precizarea a trei elemente și anume:

- structura modelului și relațiile între elementele lui
- operatorii care acționează asupra structurilor de date
- restricțiile pentru menținerea corectitudinii datelor, numite și constrângerile de integritate

Exemple de modele de date:

### Model relațional

- folosește o colecție de tabele pentru a reprezenta datele și relațiile dintre acestea
- fiecare tabelă are mai multe coloane și fiecare coloană are un nume unic

- tabelele mai poartă denumirea de relații
- este un model bazat pe înregistrări (record-uri)
- baza de date este structurată pe tipuri diferite de înregistrări cu format fix
- fiecare tabelă conține înregistrări de un anumit tip
- fiecare tip de înregistrare definește un număr fix de câmpuri sau attribute
- coloanele unei tabele corespund atributelor tipului de înregistrare

#### **Model Entitate-Relație**

- folosește o colecție de obiecte de bază, denumite entități și relațiile dintre acestea
- o entitate poate fi diferențiată de celelalte obiecte

#### **Model bazat pe obiecte**

- poate fi văzut ca o extindere a modelului Entitate-Relație cu noțiuni de încapsulare, metode și identitate de obiecte
- Model Orientat Obiect și Model Relațional-Obiect

#### **Model semi-structurat**

- permite specificarea datelor astfel încât înregistrări de același tip pot avea seturi diferite de attribute
- limbajul XML este în principal folosit pentru a reprezenta date semi-structurate

#### **Model distribuit**

- conceput pentru lucrul cu mai multe noduri de calcul
- suportă procesare distribuită a datelor

#### **Model funcțional**

- lucrează cu date multidimensionale
- folosit în aplicațiile analitice

#### **Modelul deductiv**

- folosit în sisteme expert

#### **Primele modele de date:**

#### **Modelul ierarhic**

- a fost primul model de date
- se bazează pe structuri arborescente
- relații unu-la-mulți de la nodul superior la nodurile inferioare
- relații unu-la-unu de la nodurile inferioare la nodul superior
- în cadrul fiecărui nivel, accesul se face secvențial
- nu suportă adăugarea de noi înregistrări dacă nu se cunosc părinții acestora
- dacă se șterge o înregistrare părinte, toate înregistrările dependente sunt șterse

#### **Exemple de SGBD-uri care folosesc acest model:**

- IBM Information Management System (IMS), dezvoltat în 1960
- Windows Registry, folosit în sistemele de operare Microsoft Windows

## Modelul rețea

- se bazează pe structuri în rețea
- relații mulți-la-mulți, unu-la-mulți, unu-la-unu
- mai performant decât modelul ierarhic
- accesul în cadrul nivelurilor se realizează direct

## Limbajele bazelor de date

### Limbajul de definire a datelor (Data Definition Language – DDL)

Este folosit pentru a specifica schema unei baze de date cât și o serie de proprietăți adiționale ale datelor.

Structura de stocare cât și metodele de acces folosite de un SGBD pot fi specificate printr-un set de comenzi care reprezintă un tip special de DDL, denumit **limbaj de stocare și definire a datelor (Data Storage and Definition Language – DSDL)**. Aceste comenzi definesc detaliile de implementare ale schemei unei baze de date, detalii ce sunt de obicei ascunse utilizatorilor acesteia.

DDL oferă posibilitatea de a specifica constrângeri asupra datelor. SGBD-ul verifică aceste constrângeri de fiecare dată când baza de date este modificată.

Constrângeri de integritate:

- Constrângeri de domeniu – fiecare atribut are asociat un tip de date care specifică domeniul de valori pe care le poate lua
- Constrângeri de integritate referențială – există situații în care utilizatorul dorește ca un subset de atribute dintr-o înregistrare a unei relații să se regăsească în alt subset de atribute ale aceleiași relații sau chiar a altei relații (exemplu: se poate impune ca la introducerea unui nou student, acestuia să îl fie atribuită o grupă existentă)
- Aserțiuni – constrângerile de domeniu și de integritate referențială sunt tipuri speciale de aserțiuni. Alte tipuri generale de aserțiuni pot fi specificate de utilizatori (exemplu: se poate impune ca nota unui student să nu fie negativă).
- Autorizare – impunerea unor clase diferite de drepturi pentru utilizatori: drepturi de citire, drepturi de inserare, drepturi de actualizare, drepturi de ștergere.

DDL-ul actualizează **dicționarul de date**, care mai poartă denumirea de **system catalog**. Acesta conține metadate și este stocat într-un set special de tabele care pot fi accesate și modificate doar de către sistem și nu de un utilizator obișnuit. Înainte de a citi sau modifica datele, SGBD-ul consultă dicționarul de date.

### Limbajul de manipulare a datelor (Data Manipulation Language – DML)

Este folosit pentru a interoga și actualiza datele dintr-o bază de date.

Operații suportate:

- preluarea datelor stocate în baza de date
- inserarea unor noi date în baza de date
- ștergerea datelor din baza de date
- modificarea datelor stocate în baza de date

Partea din DML care implică preluarea datelor poartă numele de **limbaj de interogare (query language)**.

Tipuri DML:

- DML procedural – utilizatorul specifică **ce** date sunt necesare și **cum** sunt preluate din baza de date
- DML declarativ (non-procedural) – utilizatorul specifică **ce** date sunt necesare **fără** a specifica cum sunt preluate din baza de date

*Exemplu:*

Presupunem că dorim să aflăm numărul total de profesori din tabela ilustrată în Fig. 1.

- Abordare procedurală

```
contor = 0
pentru fiecare înregistrare din tabela cadre_didactice
    dacă funcția este "profesor"
        contor = contor + 1
```

- Abordare declarativă (non-procedurală)

```
folosind tabela cadre_didactice calculează numărul de înregistrări în
care funcția este "profesor"
```

Limbajele DML și DDL se mai numesc și **sublimbaje** deoarece, spre deosebire de limbajele de programare de nivel înalt, nu conțin instrucțiuni pentru toate tipurile de procesare (blocuri condiționale, bucle etc.).

Multe SGBD-uri oferă posibilitatea încorporării sublimbajelor în cadrul limbajelor de programare de nivel înalt (C, C++, C#, Java etc.). În acest context, un astfel de limbaj de programare de nivel înalt se mai numește și **limbaj gazdă**. Compilarea fișierelor care conțin instrucțiuni ale sublimbajelor presupune un pas special de precompilare în care aceste instrucțiuni sunt eliminate și înlocuite cu apeluri de funcții puse la dispoziție prin intermediul unor librării specifice SGBD-ului cu care se lucrează.

*Exemplu (C):*

```
int a;
/* ... */
EXEC SQL SELECT salary INTO :a FROM Employee WHERE id=10;
/* ... */
printf("The salary is %d\n", a);
/* ... */
```

Există și limbaje de nivel înalt care oferă suport nativ pentru DML. De exemplu, Language Integrated Query (LINQ) este o componentă Microsoft .NET Framework care oferă această capacitate limbajelor .NET.

*Exemplu (C#):*

```
// interogare pentru preluarea angajaților cu salariu > 5000
var results =
    from emp in Employees
    where emp.salary > 5000
    select emp;

foreach (var emp in results)
    Console.WriteLine("name: {0}, salary: {1}", emp.name, emp.salary);
```

## Evoluție

În funcție de evoluția tehnologică, bazele de date pot fi clasificate astfel:

- naviigaționale (model ierarhic, model rețea, exemplu: Document Object Model (DOM)/JavaScript)
- relaționale (modelul relațional)
- post-relaționale (NoSQL, document-oriented)
- next generation (NewSQL)

## Roluri în cadrul sistemelor de baze de date

Administratori ai bazei de date (database administrators - DBA)

- managementul SGBD-ului
- administrarea caracteristicilor fizice ale bazei de date
- implementarea politicilor de securitate și control al integrității datelor
- menținerea unui sistem operațional
- monitorizarea sistemului și asigurarea parametrilor de performanță
- etc.

Designeri ai bazei de date (database designers)

*La nivel conceptual și logic:*

- dezvoltarea modelului de date
- identificarea datelor
- identificarea relațiilor dintre date
- identificarea constrângerilor asupra datelor (aceste constrângerile mai poartă denumirea și de **business rules**).
- etc.

La acest nivel este necesară o cunoaștere detaliată și o înțelegere profundă a datelor și a modalităților de manipulare a acestora în cadrul unei organizații.

*La nivel fizic:*

- implementarea la nivel fizic a modelului logic de date
- maparea modelului logic pe un set de tabele
- specificarea constrângerilor de integritate
- configurarea structurilor de stocare și accesare a datelor
- implementarea măsurilor de securitate pentru accesul asupra datelor
- etc.

Multe dintre atribuțiile de la acest nivel sunt dependente de SGBD-ul ales și în general există mai multe modalități de implementare. Prin urmare este necesară o cunoaștere aprofundată a caracteristicilor SGBD-ului ales cât și a avantajelor și dezavantajelor fiecărei modalități de implementare.

La nivel conceptual și logic, design-ul bazelor de date este focalizat pe **ce** date sunt necesare. La nivel fizic, focalizarea este pe **cum** sunt aceste date reprezentate și stocate.

## Analiști de sistem

Design-ul interfețelor și furnizarea specificațiilor aplicațiilor care vor fi utilizate de end-users.

## Dezvoltatori de aplicații (application developers)

Dezvoltarea aplicațiilor conform specificațiilor primite de la analiștii de sistem. Aceste aplicații se integrează cu baza de date și sunt folosite de end-users.

## Utilizatori (end-users)

### *Naïvi*

- în general nu știu de existența SGBD-ului
- accesează baza de date prin intermediul programelor furnizate de dezvoltatorii de aplicații (exemplu: casierii de la supermarket-uri folosesc o aplicație care se integrează cu scannerul de coduri de bare și cu o bază de date pentru generarea bonurilor).

### *Sofisticati*

- sunt familiari cu structura bazei de date și cu facilitățile oferite de SGBD
- folosesc limbi de interogare pentru accesarea datelor (exemplu: SQL)
- pot să își scrie propriile programe pentru accesarea bazei de date

## Istoria sistemelor de baze de date

**1832** – Semen Korsakov (rus) a folosit pentru prima oară cartele perforate și dispozitive mecanice pentru stocarea și căutarea datelor. (exemplu: homeoscop)

**1890** – Herman Hollerith (american) a dezvoltat tehnologia procesării datelor de pe cartele perforate folosind dispozitive electro-mecanice, pentru recensământul din Statele Unite care a început la 2 iunie 1890.

**1950 – 1960** – dezvoltarea benzilor magnetice pentru stocarea datelor. Procesarea datelor presupunea citirea de pe una sau mai multe benzi magnetice (sau cartele perforate) și scrierea pe o nouă bandă magnetică. Înregistrările de pe benzile/cartelele sursă trebuiau să fie în aceeași ordine. Benzile și teancurile de cartele perforate puteau fi citite doar secvențial.

**1960 – 1970** – răspândirea hard disk-urilor (HDDs) a schimbat modalitatea de procesare a datelor. La începutul acestei perioade s-au dezvoltat sistemele de date axate pe fișiere. Aceste sisteme ofereau o abordare descentralizată de stocare și procesare a datelor care prezenta o serie de neajunsuri majore, în special asupra coerenței datelor. Accesul random la datele stocate pe HDD-uri a făcut posibilă implementarea, la mijlocul perioadei, a bazelor de date de tip ierarhic și de tip rețea. Structuri de tip listă și arbore puteau fi manipulate și stocate pe HDD. Deși adresau o parte din neajunsurile sistemelor axate pe fișiere, nici aceste sisteme nu ofereau independență fizică a datelor, fiind necesară scrierea unor programe complexe pentru procesarea acestora.

**1970** – E. F. Codd care lucrat la IBM Research Laboratory, a publicat articolul care a pus bazele modelului relațional și a modalităților de interogare non-procedurală a acestuia: "A relational model of data for large shared data banks". Astfel s-au născut bazele de date relaționale.

**1970 – 1980** – Modelul relațional nu a fost inițial pus în practică din cauza performanțelor scăzute comparativ cu modelele ierarhic și rețea. Acest lucru s-a schimbat odată cu dezvoltarea proiectului Ingres (1970) în cadrul Universității Berkley și a prototipului System-R (1974) în cadrul IBM. System-R a fost conceput pentru a dovedi aspectele practice ale modelului relațional și a oferit o implementare a structurilor de date cât și a operațiilor efectuate asupra acestora. Acest lucru a dus către două dezvoltări majore ulterioare:

- dezvoltarea unui limbaj de interogare structurat, denumit SQL (Structured Query Language), care a devenit și limbajul standard pentru SGBD-urile relaționale
- dezvoltarea mai multor SGBD-uri comerciale, printre care SQL/DS (primul SGBD relațional comercial) și DB2 de la IBM și Oracle de la Oracle Corporation

**1976** – a fost propus modelul Entitate-Relație în articolul publicat de Peter Chen: "*The Entity-Relationship Model - Toward a Unified View of Data*". Acest model a devenit o parte semnificativă în cadrul metodologiilor de design a bazelor de date.

**La începutul anilor '80**, bazele de date relaționale au devenit competitive cu bazele de date care implementau modelele ierarhic și rețea. Mai mult, datorită ușurinței în utilizare, bazele de date relaționale au înlocuit treptat bazele de date ierarhice și rețea.

**1980 – 1990** – activități substanțiale de cercetare asupra bazelor de date paralele și distribuite, cât și explorarea bazelor de date orientate obiect (Object Oriented DBMS – OODBMS).

**1987** – Limbajul SQL este standardizat de către International Standards Organization (ISO). Actualizări ulterioare ale standardului: 1989, 1992 (SQL2), 1999 (SQL:1999), 2003 (SQL:2003), 2008 (SQL:2008) și 2011 (SQL:2011).

**1990 – 2000** – creșterea explozivă a World Wide Web-ului (WWW) a dus la o proliferare a sistemelor de baze de date. Acestea trebuiau să ofere rate foarte ridicate de procesare a tranzacțiilor, fiabilitate crescută, disponibilitate 24/7, cât și interfațare Web a datelor. Tot în această perioadă se dezvoltă bazele de date orientate obiect cât și bazele de date obiectual-relaționale (Object Relational DBMS - ORDBMS). De asemenea, apar și sistemele de depozitare a datelor (data warehousing systems) cât și produse de tip data-mining.

**1998** – Extensible Markup Language (XML) 1.0 ratificat de World Wide Web Consortium (W3C). Urmează integrarea XML cu SGBD-urile cât și dezvoltarea unor baze de date native XML.

**2000 – 2010** – dezvoltarea XML și a limbajului de interogare XQuery (care folosește expresii XPath) ca o nouă tehnologie de baze de date. Deși XML este folosit pe scară largă pentru schimbul de date, bazele de date relaționale rămân folosite în majoritatea aplicațiilor de baze de date de mari dimensiuni. Tot în această perioadă se reflectă și o creștere substanțială a folosirii SGBD-urilor open-source, în mod special PostgreSQL și MySQL. Spre sfârșitul perioadei s-au dezvoltat baze de date specializate cât și baze de date intens paralelizate pentru procesarea seturilor foarte mari de date. De asemenea, mai multe sisteme noi distribuite de stocare a datelor au fost dezvoltate pentru a putea face față cantității de date generate de site-uri mari precum Google, YouTube, Facebook, Amazon, Microsoft etc. O parte din aceste sisteme sunt de asemenea disponibile dezvoltatorilor de aplicații. Tot spre sfârșitul acestei perioade au fost dezvoltate noi sisteme de data-mining cât și sisteme de procesare a stream-urilor de date (Event Stream Processing).

## Avantajele și dezavantajele SGBD-urilor

### Avantaje

- control asupra redundanței datelor
- coerentă datelor
- mai multe informații pot fi generate din aceleasi date
- punerea în comun a datelor (sharing of data)
- ușurință în accesarea datelor
- posibilitatea de a defini constrângeri de integritate
- securitate și politici de securitate pentru operații diferite
- managementul accesului concurrent asupra datelor
- managementul tranzacțiilor
- impunerea standardelor
- limbaje de interogare a datelor și API-uri
- creșterea productivității
- independența fizică a datelor
- servicii de recuperare și backup
- scăderea costurilor
- etc.

### Dezavantaje

- complexitate
- necesar ridicat de memorie
- costuri de achiziție
- costuri asociate echipamentelor hardware dedicate
- costuri de migrare (portare aplicații, training-uri etc.)
- single point of failure (pentru baze de date non-distribuite)
- etc.

## Sisteme Enterprise Resource Planning (ERP)

Sistemele ERP reprezintă o categorie de software pentru managementul afacerilor. Aceste sisteme oferă o serie de aplicații integrate care sunt folosite de o organizație pentru a colecta, stoca, administra și analiza datele necesare activităților de afaceri, cum ar fi:

- planificarea producției
- planificarea costurilor
- activități de marketing
- vânzări
- managementul stocurilor
- definirea și monitorizarea lanțurilor de distribuție
- managementul plășilor
- managementul clienților
- înregistrarea și monitorizarea livrărilor
- etc.

Sistemele ERP oferă o vedere de ansamblu asupra activităților principale din cadrul unei organizații. Aceste sisteme folosesc baze de date furnizate de SGBD-uri.

Se estimează că pentru o companie din Fortune 500, în contextul ERP-urilor, costurile cu software-ul, hardware-ul și serviciile de consultanță se încadrează între 50 și 500 de milioane de dolari, depășind în general 100 de milioane de dolari. Marile companii pot cheltui între 50 și 100 de milioane de dolari pe upgrade-uri. Companiile medii, până în 1000 de angajați, se estimează că au cheltuieli între 10 și 20 de milioane de dolari.

### Avantaje

- integrare la nivel global a întregii afaceri (oferă independentă față de rate de schimb, limbă, culturi diferite, etc.)
- integrarea tuturor activităților unei organizații în cadrul același produs – proces decizional rapid
- actualizările software-ului se fac o singură dată la nivelul întregii companii
- dezvoltatorii ERP-urilor au experiență și cunoștințe necesare pentru dezvoltarea acestor sisteme
- furnizează bazele unui mediu de lucru eficient
- furnizează informații corecte în timp real
- oferă o interacțiune calitativă și extrem de eficientă cu clienții și partenerii de afaceri
- oferă o platformă de colaborare între utilizatori
- flexibilitate crescută
- interfețele aplicațiilor pot fi personalizate
- securitate crescută a datelor
- etc.

### Dezavantaje

- contracte costisitoare cu furnizorii de ERP-uri
- costuri crescute la schimbarea furnizorului ERP
- cursuri de utilizare pentru angajați
- inflexibilitate – personalizarea sau dezvoltarea/adaptarea modulelor poate fi greoaie și costisitoare
- amortizarea cheltuielilor generate cu un ERP poate dura o perioadă lungă de timp
- dificultate de integrare

etc.

### Bibliografie

- [1] Abraham Silberschat, Henry F. Korth, S. Sudarshan: **Database System Concepts (6th Edition)**, McGraw-Hill, ISBN 978-0-07-352332-3, 2011
- [2] Thomas M. Connolly, Carolyn E. Begg: **Database Systems: A Practical Approach to Design, Implementation, and Management (6th Edition)**, Pearson Education Limited, ISBN 10: 1-292-06118-9, ISBN 13: 978-1-292-06118-4, 2015
- [3] Carlos Coronel, Steven Morris: **Database Systems: Design, Implementation, and Management (11th Edition)**, ISBN-13: 978-1-285-19614-5, ISBN-10: 1-285-19614-7, Cengage Learning, 2015
- [4] Claudia Botez, Cătălin Mironeanu, Doina Buzea: **Baze de date**, Politehnium, ISBN: 978-973-621-162-1, 2009

## Design-ul bazelor de date

### Procesul de design

Structura unei baze de date este determinată în faza de design a acesteia (database design).

Pentru folosirea cu succes a bazelor de date în cadrul unei organizații, datele trebuie puse pe primul loc, înaintea aplicațiilor.

Pentru ca un sistem să funcționeze conform așteptărilor utilizatorilor, activitatea de design a bazei de date este crucială. Un sistem cu un design deficitar al bazei de date poate genera probleme de coerență a datelor, informații eronate cât și performanțe scăzute. Toate acestea pot duce la luarea unor decizii nepotrivite care pot afecta grav organizația care folosește baza de date. Pe de altă parte, un sistem cu un design bine conceput va furniza informații corecte, va facilita managementul datelor și va oferi performanțe crescute.

Un model de date de nivel înalt furnizează designer-ului bazei de date un framework conceptual în care poate specifica cerințele legate de date ale utilizatorilor cât și modalitatea de structurare a bazei de date pentru a satisface aceste cerințe.

### Componentele modelelor de date

Componentele de bază a oricărui model de date sunt:

- entități
- atribute
- relații
- constrângeri

O **entitate** reprezintă un tip particular de obiect. Fiecare entitate este unică. Exemplu de entitate: o persoană, un loc, o rută de avion, un lucru sau un eveniment pentru care sunt colectate și stocate date.

Un **atribut** reprezintă o caracteristică a unei entități. Exemplu: entitatea PERSOANĂ poate avea atributele: nume, adresă, telefon, CNP etc.

O **relație** reprezintă o asociere între entități. Relațiile sunt bidirectionale. Există trei tipuri de relații:

- **unu-la-mulți (one-to-many) – 1:M sau 1..\***. Exemplu: un profesor poate preda mai multe materii, iar o materie poate fi predată doar de un profesor. Prin urmare, relația "PROFESOR predă MATERIE" este 1:M.
- **mulți-la-mulți (many-to-many) – M:N sau \*..\***. Exemplu: un autor poate scrie mai multe cărți, iar o carte poate fi scrisă de mai mulți autori. Prin urmare, relația "AUTOR scrie CARTE" este M:N.
- **unu-la-unu (one-to-one) – 1:1 sau 1..1**. Exemplu: o persoană deține un singur pașaport, iar un pașaport este deținut de o singură persoană. Prin urmare, relația "PERSOANĂ deține PAŞAPORT" este 1:1.

O **constrângere** reprezintă o restricție care acționează asupra datelor. Constrângările ajută la păstrarea integrității datelor. Exemplu: salariul unui profesor nu trebuie să fie negativ, nota unui student trebuie să fie între 1 și 10, fiecare materie trebuie predată de un singur profesor etc.

## Reguli de business

Primul pas în identificarea corectă a entităților, atributelor, relațiilor și constrângerilor este identificarea clară a regulilor de business pentru organizația pentru care se face modelarea datelor.

O **regulă de business** este un enunț scurt, precis, lipsit de ambiguități care descrie o politică, procedură, sau principiu în cadrul unei organizații.

Regulile de business sunt folosite pentru a defini entitățile, atributele, relațiile și constrângerile. Exemplu de reguli de business:

- "un profesor poate preda mai multe materii, iar o materie este predată doar de un singur profesor"
- "numărul de studenți dintr-o grupă nu trebuie să depășească 20"
- etc.

Regulile de business descriu într-un limbaj simplu principalele caracteristici ale datelor aşa cum sunt acestea *văzute* de organizație.

Procesul de identificare și documentare al regulilor de business este esențial în design-ul bazelor de date deoarece:

- ajută la standardizarea percepției organizației asupra datelor
- sunt folosite ca și instrument de comunicare între utilizatori și designeri
- ajută designer-ul să înțeleagă natura, rolul și scopul datelor
- ajută designer-ul să înțeleagă procesele organizației
- ajută designer-ul să creeze un model corect al datelor

Există și reguli de business care nu pot fi incluse în modelul de date. Exemplu: "nici un profesor nu poate să predea mai mult de 20 ore într-o săptămână". Aceste reguli trebuie documentate și impuse la nivelul aplicațiilor software.

## Fazele procesului de design

- În faza inițială sunt analizate integral cerințele legate de date ale utilizatorilor. Rezultatul acestei faze este reprezentat printr-o specificație a cerințelor utilizatorilor.
- În faza următoare, designer-ul alege un model de date și, folosind concepții ale acestuia, transformă cerințele utilizatorilor într-o **schemă conceptuală** a bazei de date. Această schemă furnizează o privire detaliată asupra structurării datelor, a relațiilor dintre ele cât și a constrângerilor de integritate. Designer-ul verifică această schemă pentru a se asigura că toate cerințele sunt satisfăcute și că nu există conflicte între acestea. Tot la nivelul acestei faze, designer-ul elimină aspectele redundante. În acest punct, focalizarea este orientată către descrierea datelor și a relațiilor dintre ele, și nu pe stabilirea detaliilor de stocare fizică a acestora.
- În a treia fază, designer-ul, în baza schemei conceptuale, construiește **design-ul logic** al bazei de date și stabilește toate atributele entităților participante. Cu alte cuvinte, designer-ul mapează elementele schemei conceptuale pe modelul de date utilizat de SGBD-ul care urmează a fi folosit, obținând astfel **schema logică** a bazei de date.
- În ultima fază, designer-ul folosește schema logică a bazei de date pentru a specifica elemente care aparțin de **design-ul fizic** al bazei de date (tipuri de date folosite, modalitate de organizare a fișierelor, structuri de stocare interne ale SGBD-ului etc.).

## Modelul Entitate-Relație

Modelul relațional, deși oferă o alternativă mult îmbunătățită față de modelele ierarhic și rețea, nu posedă instrumentele necesare pentru a putea fi eficient în design-ul unei baze de date. Deoarece design-ul structurilor de date este făcut mult mai ușor folosind instrumente grafice decât text, designerii bazelor de date preferă o abordare grafică prin care entitățile și relațiile dintre acestea sunt reprezentate. Astfel, modelul Entitate-Relație (ER model) a devenit un standard larg acceptat pentru modelarea datelor.

Modelul ER a fost pentru prima oară propus de Peter Chen în anul 1976. Modul de reprezentare grafică a entităților și a relațiilor dintre ele a crescut rapid în popularitate deoarece complementă în mod natural concepțile modelului relațional. Astfel, cele două modele, modelul relațional și modelul ER, s-au combinat într-o fundație solidă pentru design-ul bazelor de date. Modelul ER este reprezentat grafic prin diagrame entitate-relație.

### Notații ale modelului ER

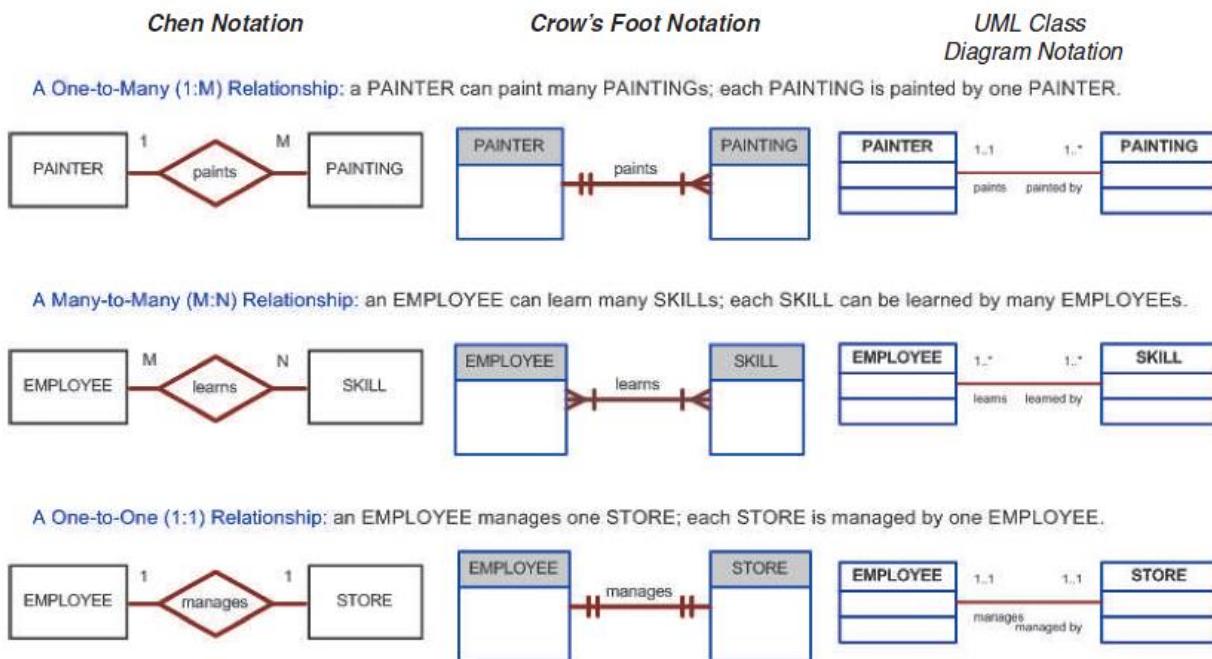


Fig. 1. Notații ale modelului ER [3]

## Modelarea datelor

- un model de date trebuie să ofere un anumit grad de simplicitate conceptuală fără a compromite semantica datelor
- modelul de date trebuie să fie clar, concis și fără ambiguități
- modelul de date trebuie să fie corect, complet și să reflecte îndeaproape problema pe care o modeleză
- reprezentarea regulilor de business trebuie să fie în concordanță cu constrângerile de integritate definite la nivelul modelului de date

## Grade de abstractizare a datelor

În anul 1975, American National Standards Institute (ANSI) Standards Planning and Requirements Committee (SPARC) a definit un framework pentru modelarea datelor bazat pe grade de abstractizare a datelor. Arhitectura ANSI/SPARC rezultată prezintă trei niveluri de abstractizare a datelor: extern, conceptual și intern.

Fig. 2 prezintă un nivel adițional, nivelul fizic, pentru a ilustra în mod explicit detalii fizice de stocare ale modelului intern.

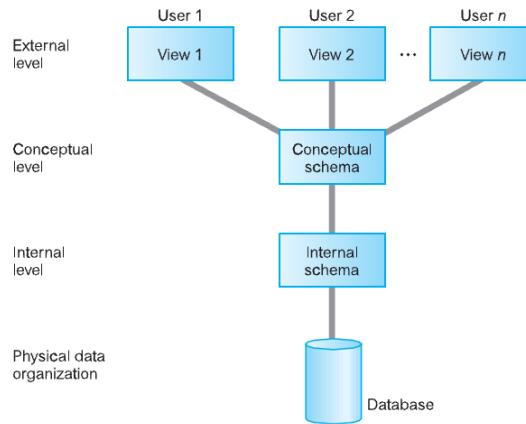


Fig. 2. ANSI/SPARC: Niveluri de abstractizare a datelor [2]

Deși arhitectura ANSI/SPARC nu a devenit niciodată standard, aceasta oferă o bază de înțelegere a unor serii de concepte din cadrul SGBD-urilor.

Această arhitectură definește trei niveluri distincte la care datele pot fi descrise, fiecare nivel având propria schemă. La cel mai înalt nivel de abstractizare avem **schemele externe**, denumite și **subscheme**, care corespund unor view-uri diferite asupra datelor. La nivelul conceptual avem **schema conceptuală** ce descrie toate entitățile, atributele acestora, relațiile dintre acestea cât și constrângările de integritate. La cel mai scăzut nivel de abstractizare avem **schema internă** care descrie complet modelul intern de date: definirea structurilor de stocare și a modalității de reprezentare a datelor, a structurilor de indexare a datelor etc.

*Utilizatorii percep datele la nivelul extern. SGBD-ul percep datele la nivelul intern (nivelul la care structurile de date pentru stocarea efectivă a datelor sunt definite și implementate). Nivelul conceptual oferă independență și mapările necesare între nivelurile extern și intern.*

Obiectivul acestei arhitecturi pe trei niveluri este de a separa view-urile externe (de la nivelul extern) asupra bazei de date unele de altele. Un alt obiectiv este de a le separa de modalitatea în care baza de date este reprezentată fizic. Aceste separări asigură faptul că:

- fiecare utilizator trebuie să poată accesa aceleași date ca și ceilalți utilizatori, posibil printr-un view diferit față de aceștia
- view-ul unui utilizator trebuie să poată fi modificat independent, fără a afecta view-urile altor utilizatori

- utilizatorii nu trebuie să interacționeze cu mecanismele de reprezentare ale bazei de date de la nivelul fizic. Interacțiunea utilizatorilor cu baza de date trebuie să fie independentă de modalitatea de stocare a datelor.
- administratorul bazei de date (DBA) trebuie să poată schimba structurile de stocare a bazei de date fără a afecta view-urile utilizatorilor
- structura internă a bazei de date nu trebuie să fie afectată de schimbările de la nivelul fizic ale mediului de stocare (exemplu: trecerea de la stocare pe HDD la stocare în rețea sau pe alt suport media)
- administratorul bazei de date (DBA) trebuie să poată face schimbări asupra modelului conceptual fără a afecta utilizatorii sau afectând doar un număr restrâns de utilizatori

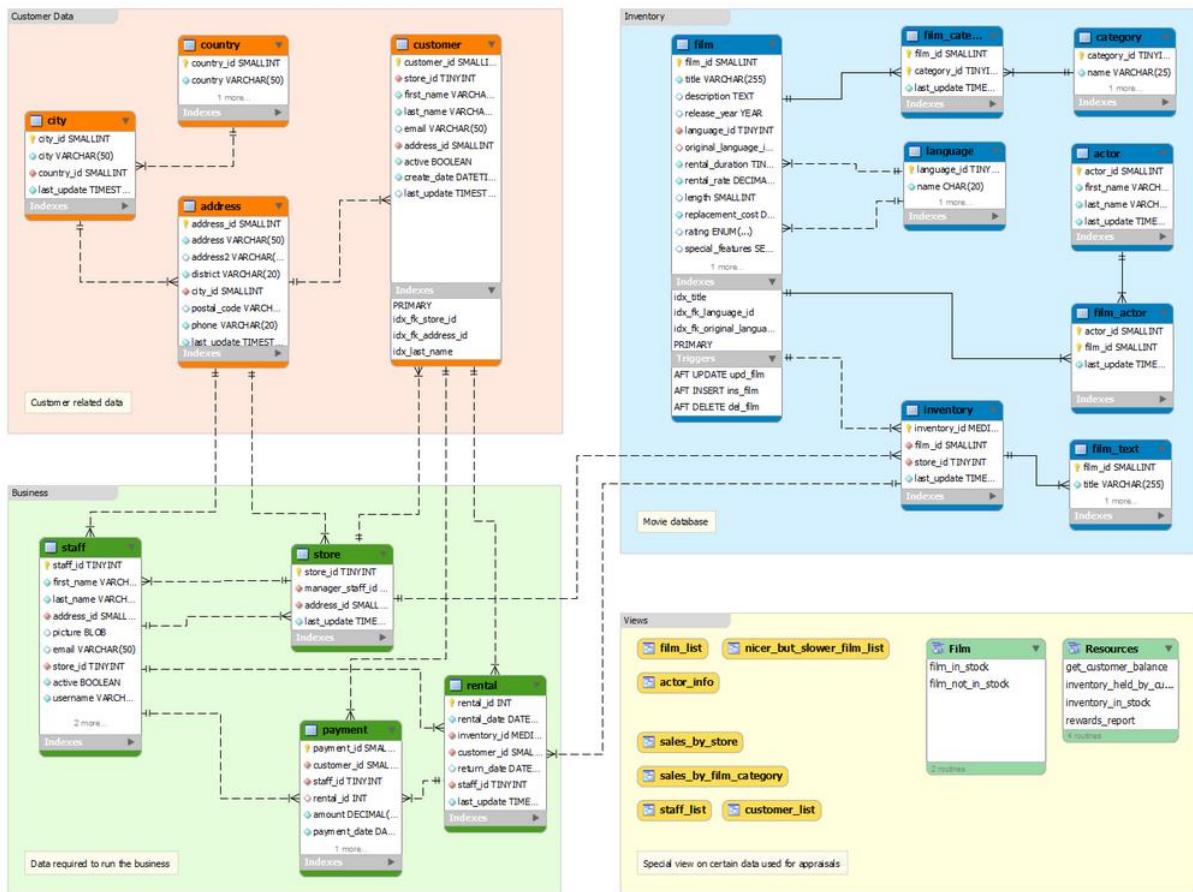


Fig. 3. Model de date Sakila (Oracle/MySQL) – diagrame ER la nivelurile extern și conceptual

## Nivelul extern

Modelul extern reprezintă *vederea* utilizatorilor asupra bazei de date. La acest nivel sunt descrise secțiunile bazei de date care sunt relevante utilizatorilor/grupurilor de utilizatori.

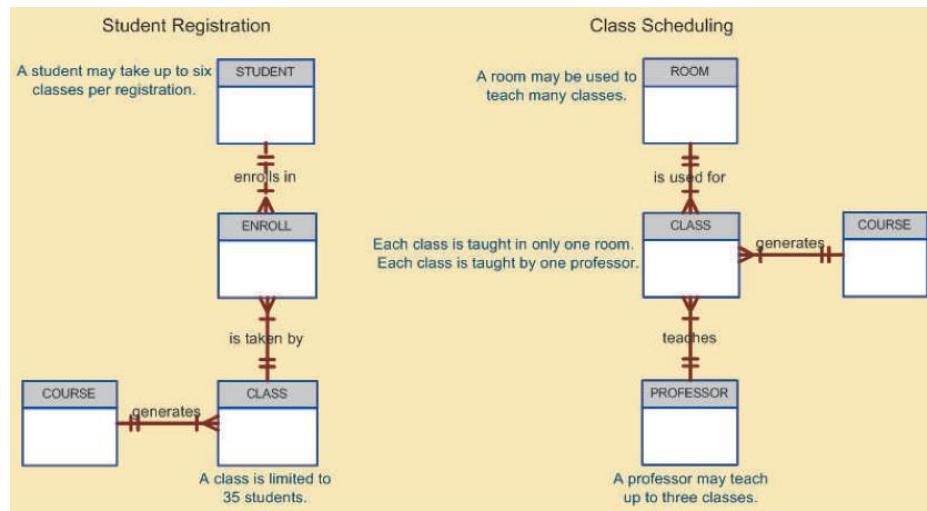


Fig. 4. Modele/view-uri externe – diagrame ER [3]

Utilizatorii folosesc în general aplicații scrise de programatori pentru a manipula datele din baza de date și pentru a extrage informații. De obicei, în cadrul unei organizații, utilizatorii lucrează în cadrul unor unități distincte/departamente cu specific propriu (exemplu: contabilitate, vânzari, resurse umane etc.). Fiecare departament, în funcție de specific, folosește programe diferite pentru accesarea și manipularea unui subset de date din baza de date. Prin urmare, seturile de date pe care lucrează utilizatorii din cadrul acestor departamente sunt în general diferite. Fiecare astfel de set de date este reprezentat de un model/view extern.

În cadrul procesului de modelare a datelor sunt folosite diagrame ER pentru reprezentarea view-urilor externe. O reprezentare specifică a unui view extern poartă denumirea de **schemă externă**.

Folosirea view-urilor externe ce reprezintă subseturi de date din baza de date prezintă câteva avantaje importante:

- identificarea datelor specifice necesare unui anumit departament se poate face cu ușurință
- ușurează munca designerului modelului de date: acuratețea modelului poate fi evaluată prin verificarea suportului modelului pentru toate procesele definite de modelele externe, cât și a constrângerilor de integritate prezente la nivelul acestora
- ajută la implementarea politicilor de securitate: afectarea întregii baze de date este mult mai dificilă deoarece fiecare departament lucrează doar pe un subset de date
- dezvoltarea aplicațiilor de către programatori este mult mai simplă deoarece nu presupune același grad de complexitate ca atunci când se lucrează pe un set complet de date

## Nivelul conceptual

Modelul conceptual descrie **ce** date sunt stocate în baza de date și care sunt relațiile și constrângerile de integritate care acționează asupra acestora.

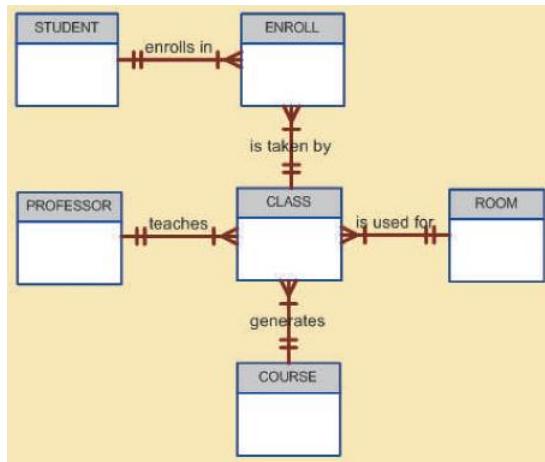


Fig. 5. Model conceptual – diagramă ER [3]

Nivelul conceptual oferă o vedere globală și furnizează structura logică a întregii baze de date a întregii organizații. Modelul conceptual integrează toate view-urile externe (entități, relații, constrângerii, procese) într-un singur view global asupra tuturor datelor din organizație. O reprezentare specifică a modelului conceptual poartă denumirea de **schemă conceptuală**. Acest model oferă o descriere la nivel înalt a datelor, fără a specifica detalii de implementare/stocare a acestora.

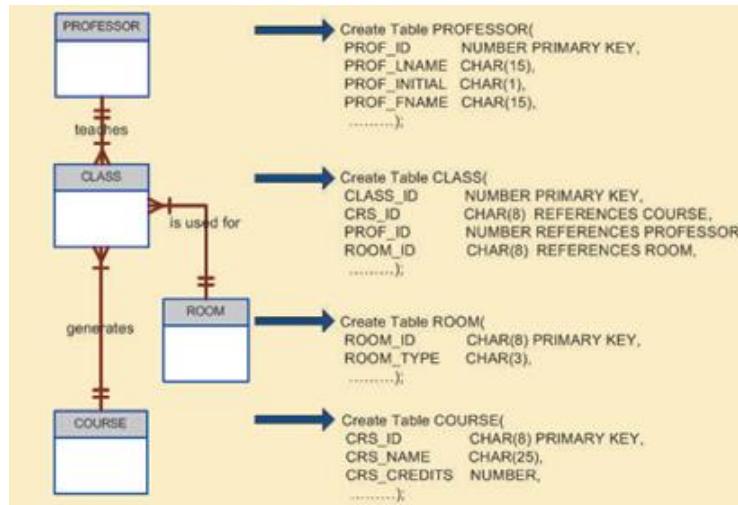


Fig. 6. Model conceptual – mapare relațională [3]

Cel mai folosit model la nivelurile conceptual și extern este modelul ER. Diagramele ER sunt folosite pentru a reprezenta grafic modelul conceptual.

Avantajele modelului ER:

- în cadrul nivelului conceptual, oferă o vedere globală asupra datelor
- este independent de hardware și software

Prin **independență software** se înțelege că modelul ER nu depinde de SGBD-ul care este folosit pentru să implemente. Prin **independență hardware** se înțelege că modelul ER nu depinde de hardware-ul folosit pentru implementarea acestuia. Prin urmare, potențialele modificări aduse software-ului SGBD-ului sau hardware-ului folosit nu vor avea nici un efect asupra modelului ER de la nivelurile conceptual și extern.

Modelul conceptual înglobează:

- toate entitățile și atributele acestora, cât și relațiile dintre ele
- constrângerile de integritate asupra datelor
- informații semantice asupra datelor
- politici de securitate

Nivelul conceptual acționează ca o bază pentru fiecare view extern: datele accesate de un utilizator prin intermediul view-urilor externe trebuie să fie conținute sau derive din modelul conceptual. Mai mult, acest model nu trebuie să conțină detalii de stocare a datelor. De exemplu, descrierea unei entități trebuie să conțină doar tipurile de date ale atributelor sale (integer, varchar etc.) și lungimea acestora (numărul maxim de decimale sau numărul maxim de caractere etc.), nu și detalii de stocare la nivel fizic (număr de octeți, ordinea octetilor – big endian, little endian, detalii de fragmentare a datelor etc.).

#### Nivelul intern

Modelul intern descrie **cum** sunt stocate datele în baza de date. Acest model este o reprezentare a bazei de date aşa cum este aceasta *văzută* de SGBD. O reprezentare specifică a modelului intern poartă denumirea de **schemă internă**.

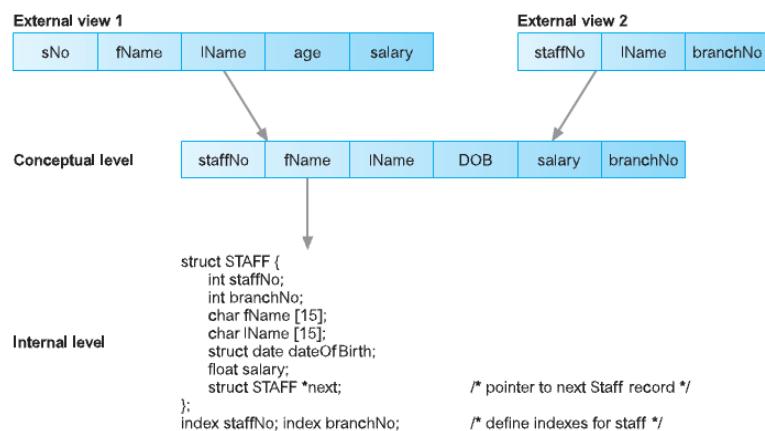


Fig. 7. Model intern [3]

Modelul intern descrie structurile de date și organizarea fișierelor folosite pentru a stoca datele. Acest model comunică cu sistemul de operare pentru a accesa/manipula datele și structurile de date de pe suportul extern.

Atribuțiile modelului intern:

- alocarea spațiului necesar pentru stocarea datelor și a informațiilor de indexare
- descrierea detaliată a înregistrărilor și a primitivelor de stocare aferente acestora
- plasarea înregistrărilor
- compresia și criptarea datelor

În contextul primelor modele de date, designer-ul era forțat să specifice detaliile fizice de stocare a datelor. Spre deosebire de acestea, modelul relațional pune accentul pe nivelul logic și nu necesită specificarea detaliilor fizice. Totuși, pentru creșterea performanțelor, este posibilă specificarea unor detalii fizice (de nivel înalt) chiar și în cadrul modelului relațional.

### Nivelul fizic

Acest nivel se află sub nivelul intern și este administrat de sistemul de operare la indicațiile SGBD-ului. La acest nivel se regăsesc detalii de stocare cunoscute doar de sistemul de operare (exemplu: modalitatea de stocare a fișierelor/înregistrărilor pe suportul extern în funcție de sistemul de fișiere folosit de sistemul de operare).

### Maparea schemelor

În cadrul arhitecturii ANSI/SPARC, pentru o bază de date există una sau mai multe scheme externe, o singură schemă conceptuală și o singură schemă internă.

SGBD-ul este responsabil pentru toate mapările dintre aceste tipuri de scheme (Fig. 8). Aceasta trebuie să verifice schemele din punct de vedere al coerenței: trebuie să verifice dacă fiecare schemă externă este derivabilă din schema conceptuală și să folosească informațiile din schema conceptuală pentru a mapa schemele externe pe schema internă.

Schema conceptuală este mapată pe schema internă prin **maparea conceptuală/internă**. Această mapare permite găsirea înregistrărilor pe suportul fizic corespunzătoare înregistrărilor logice de la nivelul conceptual împreună cu toate constrângerile aferente.

Fiecare schemă externă este mapată pe schema conceptuală prin **maparea externă/conceptuală**. Această mapare permite corelarea view-urilor utilizatorilor cu secțiunile corespunzătoare de la nivelul conceptual.

### Independența datelor

Un obiectiv major al arhitecturii pe trei niveluri de abstractizare ANSI/SPARC este acela de a oferi **independență datelor**. Acest lucru se traduce prin imunitatea nivelurilor superioare la schimbările produse în nivelurile inferioare.

*Există două tipuri de independență a datelor: logică și fizică.*

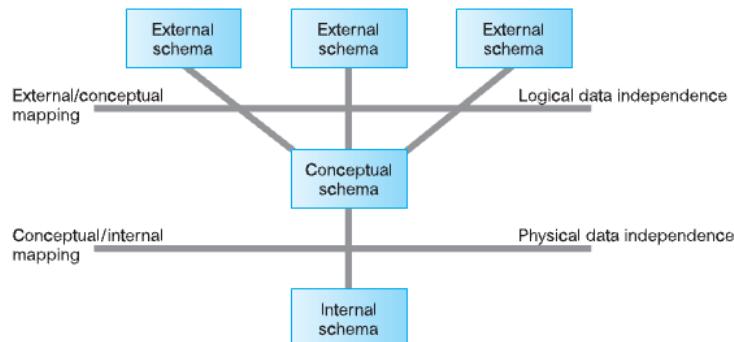


Fig. 8. ANSI/SPARC: maparea schemelor și independența datelor [2]

**Independența logică a datelor** presupune imunitatea schemelor externe la schimbările produse în cadrul schemei conceptuale.

Schimbările din cadrul schemei conceptuale (adăugarea/ștergerea de entități, attribute, relații între date etc.) nu trebuie să afecteze schemele externe și nici aplicațiile existente. Cel mult sunt afectate doar schemele externe și/sau aplicațiile unui utilizator/grup de utilizatori pentru care s-au făcut explicit aceste schimbări.

**Independența fizică a datelor** presupune imunitatea schemei conceptuale la schimbările produse în cadrul schemei interne.

Schimbările din cadrul schemei interne (modificarea organizării datelor în fișiere, modificarea structurilor de stocare, folosirea altor medii de stocare, modificarea algoritmilor de indexare etc.) nu trebuie să afecteze schema conceptuală și nici schemele externe. Din punct de vedere al utilizatorilor, singurul efect care ar putea fi percepțut este modificarea performanțelor SGBD-ului.

## Bibliografie

- [1] Abraham Silberschat, Henry F. Korth, S. Sudarshan: ***Database System Concepts (6th Edition)***, McGraw-Hill, ISBN 978-0-07-352332-3, 2011
- [2] Thomas M. Connolly, Carolyn E. Begg: ***Database Systems: A Practical Approach to Design, Implementation, and Management (6th Edition)***, Pearson Education Limited, ISBN 10: 1-292-06118-9, ISBN 13: 978-1-292-06118-4, 2015
- [3] Carlos Coronel, Steven Morris: ***Database Systems: Design, Implementation, and Management (11th Edition)***, ISBN-13: 978-1-285-19614-5, ISBN-10: 1-285-19614-7, Cengage Learning, 2015
- [4] Claudiu Botez, Cătălin Mironeanu, Doina Buzea: ***Baze de date***, Politehnium, ISBN: 978-973-621-162-1, 2009

## Modelul relațional

### Introducere

**SGBDR**-urile (Sistemele de gestiune a bazelor de date relaționale) / **RDBMS**-urile (Relational Database Management Systems) au o poziție dominantă în contextul software-urilor de procesare a datelor. Acest tip de sisteme reprezintă a doua generație de **SGBD**-uri (Sisteme de gestiune a bazelor de date) / **DBMS**-uri (Database Management Systems) având la bază modelul de date relațional propus de E. F. Codd în anul 1970, în lucrarea "*A relational model of data for large shared data banks*".

În cadrul modelului relațional toate datele sunt structurate logic în cadrul relațiilor (tabelelor). Această modalitate logică de organizare a datelor are la bază o fundamentare teoretică riguroasă care nu se regăsea în prima generație de SGBD-uri (ierarhice și rețea).

Modelul relațional are la bază logica predicatorilor și teoria seturilor. Prin urmare, acest model dispune de trei componente bine definite:

- o structură logică reprezentată de relații
- un set de reguli de integritate care asigură faptul că datele sunt și rămân coerente
- un set de operații care descriu cum sunt manipulate datele

### Obiective

Obiectivele modelului relațional:

- furnizarea unui grad ridicat de independență a datelor: aplicațiile nu trebuie să fie afectate de modificările aduse reprezentării interne a datelor (schimbări în modalitatea de organizare a fișierelor, a ordinii înregistrărilor, a căilor de acces etc.).
- furnizarea unor concepe de bază pentru adresarea semantică și coerentă a datelor, cât și pentru problemele de redundanță a acestora. În acest context, Codd a introdus conceptul de relații normalizate.
- dezvoltarea limbajelor de manipulare a datelor orientate pe seturi.

### Definiții

O **relație** reprezintă un concept matematic folosit pentru a defini structura logică a datelor în cadrul modelului relațional. O **relație** poate fi percepută ca o **tabelă**. Codd a folosit termenii **relație** și **tabelă** ca fiind sinonimi. O **relație/tabelă** reprezintă o structură bi-dimensională, compusă din linii și coloane.

Termenul **relație** provine din teoria seturilor. În contextul în care modelul relațional este unul dintre modelele care folosește perechi de atribute pentru a stabili relații (cu sensul de legături) între date, mulți cred în mod greșit că termenul **relație** se referă la relațiile/legăturile dintre date.

În cadrul unui SGBDR, baza de date este percepută de utilizatori ca un set de tabele/relații.

Un **atribut** reprezintă o coloană a unei relații/tabele, iar fiecare coloană are un nume unic. Atributele pot apărea în cadrul unei relații în orice ordine, fără a schimba sensul acesteia.

**Domeniul unui atribut** reprezintă un set de valori pe care le poate lua acel atribut. Prin urmare, fiecare coloană a unei tabele are un set specific de valori date de domeniul atributului.

În cazul mai multor atribute, domeniile acestora pot fi diferite sau pot reprezenta același domeniu.

Conceptul de **domeniu** este foarte important deoarece oferă posibilitatea de a specifica într-un singur loc tipul de date și setul de valori pe care un atribut le poate lua. Prin urmare, SGBDR-ul dispune de mai multe informații cu privire la tipul de date al atributelor și poate detecta operații semantice incorecte (exemplu: adunarea atributului *salariu*, tip numeric, cu atributul *nume*, tip sir de caractere).

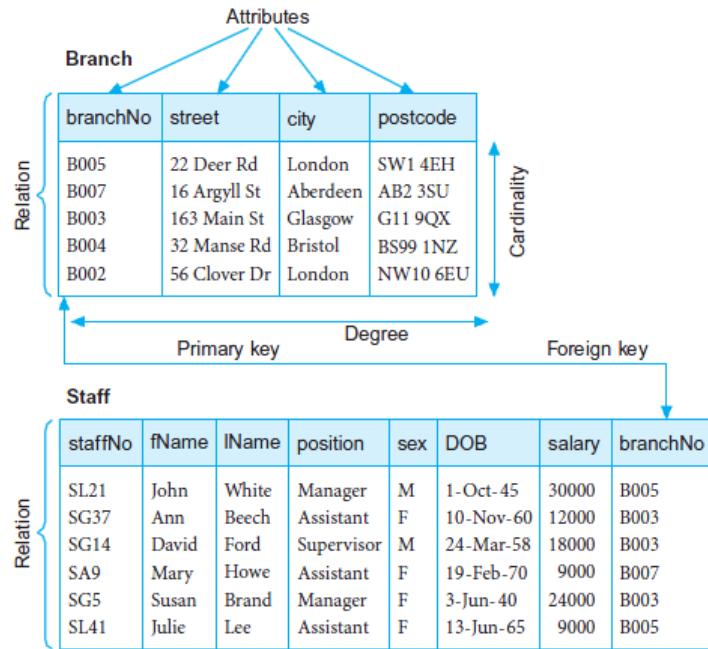


Fig. 1. Concepte ale modelului relational [2]

Attribute	Domain Name	Meaning	Domain Definition
branchNo	BranchNumbers	The set of all possible branch numbers	character: size 4, range B001–B999
street	StreetNames	The set of all street names in Britain	character: size 25
city	CityNames	The set of all city names in Britain	character: size 15
postcode	Postcodes	The set of all postcodes in Britain	character: size 8
sex	Sex	The sex of a person	character: size 1, value M or F
DOB	DatesOfBirth	Possible values of staff birth dates	date, range from 1-Jan-20, format dd-mmm-yy
salary	Salaries	Possible values of staff salaries	monetary: 7 digits, range 6000.00–40000.00

Fig. 2. Exemplu de domenii de atribute [2]

O **tuplă** reprezintă o linie din cadrul unei relații/tabele. O relație/tabelă este constituită din linii sau tuple. Fiecare tuplă conține valori pentru fiecare dintre attributele relației. Tuplele pot apărea în cadrul unei relații în orice ordine, fără a schimba sensul acestia.

Prin urmare, atât tuplele (liniile) cât și attributele (coloanele) unei relații (tabele) pot apărea în orice ordine, fără a modifica în vreun fel sensul datelor.

O **celulă** reprezintă intersecția dintre o linie și o coloană. Valoarea acestei celule este valoarea atributului corespunzător coloanei în contextul tuplei corespunzătoare liniei.

Structura unei relații împreună cu specificarea domeniilor atributelor și a restricțiilor asupra datelor poartă denumirea de **schemă** a relației, sau **intensie**. În general, aceasta nu se modifică în timp.

Conținutul relației, alcătuit din tuple, poartă denumirea de **extensie** (sau **stare**) a relației. În general, aceasta suferă modificări dese în timp.

**Gradul unei relații** este dat de numărul de atrbute conținute de aceasta. O relație cu un atrbut se numește **relație unară**, și are gradul unu. O relație cu două atrbute se numește **relație binară**, și are gradul doi. O relație cu n atrbute se numește **relație n-ară**, și are gradul n. *Gradul unei relații este o proprietate a intensiei relației.*

**Cardinalitatea unei relații** este dată de numărul de tuple conținute de aceasta. *Cardinalitatea unei relații este o proprietate a extensiei relației.*

O **bază de date relațională** este o colecție de relații, fiecare dintre acestea având un nume unic.

Termeni formalni	Alternativa 1	Alternativa 2
relație	tabelă	fișier
tuplă	linie	înregistrare
atrbut	coloană	câmp

Fig. 3. Terminologie alternativă

## Relații matematice

Pentru a înțelege conceptul de relație în cadrul modelului relațional, trebuie să înțelegem partea matematică din spatele acestui concept.

Fie n seturi  $S_1, S_2, \dots, S_n$ . **Produsul cartezian** al acestor n seturi, notat cu  $S_1 \times S_2 \times \dots \times S_n$  (sau cu  $\prod_{i=1}^n S_i$ ), este compus din setul tuturor perechilor ordonate în care primul element este din  $S_1$ , al doilea element este din  $S_2, \dots$ , al n-lea element este din  $S_n$ .

$$S_1 \times S_2 \times \dots \times S_n = \{ (s_1, s_2, \dots, s_n) \mid s_1 \in S_1 \wedge s_2 \in S_2 \wedge \dots \wedge s_n \in S_n \}$$

O **relație** reprezintă un subset (orice subset) al produsului cartezian. Prin urmare, în cazul a n seturi, o relație reprezintă un subset (orice subset) de **n-tuple** din setul complet de n-tuple generat de produsul cartezian.

Exemplu pentru două seturi:

$$S_1 = \{1, 3\}, S_2 = \{2, 3, 4\}$$

$$S_1 \times S_2 = \{(1, 2), (1, 3), (1, 4), (3, 2), (3, 3), (3, 4)\}$$

$$\text{relația } r_1 = \{(1, 3), (3, 2)\}, \text{ relația } r_2 = \{(1, 2), (1, 4), (3, 3)\} \text{ etc.}$$

O relație poate fi specificată prin-un set de **condiții de selecție**. De exemplu, putem selecta toate perechile pentru care al doilea element are valoarea 4. Rezultă relația:  $r = \{(1, 4), (3, 4)\}$ . Această relație poate fi definită astfel:

$$r = \{(x, y) \mid x \in S_1 \wedge y \in S_2 \wedge y = 4\}$$

Un alt exemplu: selecția tuturor perechilor pentru care al doilea element este cu 1 mai mare decât primul element. Rezultă relația  $r = \{(1, 2), (3, 4)\}$ . Această relație poate fi definită astfel:

$$r = \{(x, y) \mid x \in S_1 \wedge y \in S_2 \wedge y = x + 1\}$$

Un alt exemplu: selecția tuturor perechilor pentru care primul element este egal cu al doilea element. Rezultă relația  $r = \{(3, 3)\}$ . Această relație poate fi definită astfel:

$$r = \{(x, y) \mid x \in S_1 \wedge y \in S_2 \wedge x = y\}$$

Exemplu pentru trei seturi:

$$S_1 = \{1, 9\}, S_2 = \{2, 5, 7\}, S_3 = \{1, 4\}$$

$$\begin{aligned} S_1 \times S_2 \times S_3 = & \{(1, 2, 1), (1, 2, 4), (1, 5, 1), (1, 5, 4), (1, 7, 1), (1, 7, 4), \\ & (9, 2, 1), (9, 2, 4), (9, 5, 1), (9, 5, 4), (9, 7, 1), (9, 7, 4)\} \end{aligned}$$

relația  $r_1 = \{(1, 5, 1), (1, 7, 4), (9, 7, 1), (9, 7, 4)\}$ , relația  $r_2 = \{(1, 2, 1), (9, 2, 4)\}$  etc.

Selecția tuturor tuplelor pentru care primul și ultimul element au aceeași valoare. Rezultă relația:  $r = \{(1, 2, 1), (1, 5, 1), (1, 7, 1)\}$ . Această relație poate fi definită astfel:

$$r = \{(x, y, z) \mid x \in S_1 \wedge y \in S_2 \wedge z \in S_3 \wedge x = z\}$$

Un alt exemplu: selecția tuturor tuplelor strict descrescătoare. Rezultă relația:  $r = \{(9, 2, 1), (9, 5, 1), (9, 5, 4), (9, 7, 1), (9, 7, 4)\}$ . Această relație poate fi definită astfel:

$$r = \{(x, y, z) \mid x \in S_1 \wedge y \in S_2 \wedge z \in S_3 \wedge x > y \wedge y > z\}$$

### Relații în modelul relațional

În contextul matematic discutat, **schema unei relații** este definită de un set de perechi compuse din numele atributului și domeniul atributului.

Fie atributele  $A_1, A_2, \dots, A_n$  și domeniile aferente acestora  $D_1, D_2, \dots, D_n$ . Schema relației este:

$$R = \{A_1:D_1, A_2:D_2, \dots, A_n:D_n\}.$$

O relație  $r$  definită de schema relației  $R$  este notată cu  $r(R)$  și reprezintă un set de  $n$ -tuple ce conțin mapări de la numele atributelor la valori din domeniile corespunzătoare acestora. O astfel de  $n$ -tuple are forma:

$$\{(A_1:d_1, A_2:d_2, \dots, A_n:d_n) \mid d_1 \in D_1 \wedge d_2 \in D_2 \wedge \dots \wedge d_n \in D_n\}$$

Fiecare element al  $n$ -tuplei constă din numele atributului și valoarea acestuia. Când această relație este reprezentată ca o tabelă, aşa cum este cazul modelului relațional, numele atributelor  $A_1, A_2, \dots, A_n$  sunt trecute în header-ele coloanelor, iar  $n$ -tuplele sunt trecute în liniile tabelei sunt forma  $(d_1, d_2, \dots, d_n)$ .

Prin urmare, putem vedea o relație din modelul relațional ca un subset (orică subset) al produsului cartezian al domeniilor atributelor. O tabelă este o reprezentare a acestei relații.

**Schema unei baze de date relaționale** reprezintă un set de scheme de relații, fiecare având un nume unic. Dacă  $R_1, R_2, \dots, R_n$  sunt schemele de relații din baza de date relațională, atunci schema acesteia este:

$$R = \{R_1, R_2, \dots, R_n\}$$

## Proprietățile relațiilor

O relație are următoarele proprietăți:

- are un nume unic în cadrul schemei bazei de date relaționale
- fiecare celulă conține o singură valoare atomică corespunzătoare domeniului atributului
- fiecare atribut are un nume unic în cadrul schemei relației
- toate valorile unui atribut fac parte din același domeniu, domeniul atributului
- fiecare tuplu este unic: nu există duplicate
- ordinea atributelor nu este semnificativă
- ordinea tuplelor nu este semnificativă

## Chei relaționale

O **supercheie (superkey)** reprezintă un atribut sau un set de attribute care identifică unic o tuplă în cadrul unei relații. O supercheie poate conține un număr mai mare de attribute decât este necesar pentru a putea identifica unic o tuplă.

O **cheie candidat (candidate key)** este o supercheie pentru care nici un subset strict de attribute ale acesteia nu reprezintă o supercheie.

Fie K o cheie candidat pentru relația r(R). Aceasta are următoarele proprietăți:

- unicitate – pentru fiecare tuplă din r(R), valorile cheii candidat K identifică unic acea tuplă
- ireductibilitate – nu există nici un subset strict al lui K care să aibă proprietatea de unicitate

O relație poate avea mai multe chei candidat.

Dacă o cheie conține două sau mai multe attribute atunci aceasta se mai numește **cheie compusă (composite key)**.

CititorID	CartelID	Dată_împrumut
1	30	01/02/2016
4	30	12/01/2016
5	10	05/03/2016
8	20	04/08/2016

Fig. 4. Împrumuturi bibliotecă

Tabela din Fig. 4 conține date cu privire la împrumuturile făcute de cititori în cadrul unei biblioteci. Deși atributul CititorID pare că identifică unic tuplele, în semantica datelor de împrumut, acest lucru nu se menține. Este foarte posibil ca următoarea tuplă introdusă în tabelă să conțină o valoare pentru CititorID care se repetă.

Dacă politica bibliotecii **nu** lasă cititorii să împrumute aceeași carte de mai multe ori, atunci perechea <CititorID, CartelID> este suficientă pentru a identifica în mod unic o tuplă și devine cheie candidat compusă. Pe de altă parte, dacă politica bibliotecii permite cititorilor să împrumute aceeași carte de mai multe ori, atunci perechea <CititorID, CartelID> nu va mai identifica unic tuplele și nu va mai fi cheie candidat. În acest caz, tripla <CititorID, CartelID, Dată\_împrumut> va identifica unic tuplele și va reprezenta cheia candidat compusă.

Conținutul unei relații la un moment dat **nu** poate fi folosit pentru a identifica faptul că un atribut sau un set de attribute formează o cheie candidat. Simplul fapt că în cadrul relației, *la un moment dat*, nu există duplicate pentru aceste attribute nu garantează proprietatea de unicitate a acestora. Pe de altă parte, prezența dupliilor la un moment dat exclude clar posibilitatea de a forma o cheie candidat.

Prin urmare, identificarea cheilor candidat presupune cunoașterea semantică datelor care urmează a fi stocate în tabelă. Aceasta este singura modalitate de a afirma cu certitudine că un atribut sau un set de attribute formează o cheie candidat.

**O cheie primară (primary key)** este una dintre cheile candidat care este aleasă pentru a reprezenta unic tuplele din cadrul unei relații.

Deoarece în cadrul unei relații nu există tuple dupli, este întotdeauna posibilă identificarea unică a acestora folosind un atribut sau set de attribute. În cel mai rău caz, toate attributele pot fi folosite pentru identificarea unică a tuplelor.

Cheile candidat care nu sunt alese pentru a fi cheia primară, poartă numele de **chei alternative (alternate keys)**.

**O cheie străină (foreign key)** reprezintă un atribut sau un set de attribute din cadrul unei relații care referă (are/au aceeași valoare) o cheie candidat a unei alte relații (posibil aceeași relație).

Când un atribut apare în mai multe relații, prezența acestuia denotă de obicei o legătură între tuplele acestor relații.

**O cheie secundară (secondary key)** reprezintă o cheie folosită strict pentru regăsirea datelor (exemplu: un student poate nu a reținut numărul matricol folosit ca și cheie primară; în acest caz, numele și prenumele acestuia pot fi folosite ca și cheie secundară pentru regăsirea datelor sale). *O cheie secundară poate să nu identifice în mod unic o tuplă*. Prin urmare, o cheie secundară este cu atât mai eficientă cu cât restrâne cât mai mult lista de rezultate găsite.

## Constrângeri de integritate

### Constrângeri de domeniu

Deoarece fiecărui atribut îi este asociat un domeniu, valorile pe care le pot lua attributele în cadrul relațiilor sunt restricționate la un set bine determinat de valori din cadrul domeniilor asociate. Aceste restricții poartă denumirea de **constrângeri de domeniu**.

### Conceptul null

Conceptul **null** este folosit pentru a specifica faptul că:

- valoarea unui atribut nu este cunoscută
- valoarea unui atribut nu se aplică contextului curent
- valoarea unui atribut nu a fost încă determinată

Conceptul null reprezintă o modalitate de a modela datele incomplete, indisponibile sau excepțiile care pot apărea în cadrul modelării datelor. "Valoarea" null **nu** reprezintă același lucru cu valoarea numerică zero sau cu sirul de caractere vid sau care conține doar spații.

"Valoarea" null reprezintă absența valorii. Prin urmare, "valorile" null sunt tratate diferit de celelalte valori. Deoarece conceptul de null nu reprezintă o valoare ci absența acesteia, termenul de "valoare nulă" este incorect.

CititorID	CarteID	Dată_împrumut	Dată restituire	Comentarii
1	30	01/02/2016	10/02/2016	NULL
4	30	12/01/2016	03/03/2016	Lipsă copertă
5	10	05/03/2016	NULL	NULL
8	20	04/08/2016	NULL	Lipsă pagina 10

Fig. 5. Împrumuturi bibliotecă (varianta extinsă)

#### Constrângerea de integritate a entităților

Această constrângere de integritate se aplică cheilor primare (primary keys).

**Constrângerea de integritate a entităților** presupune faptul că în cadrul unei relații/tabele, nici un atribut al cheii primare nu poate fi null. Această constrângere asigură faptul că orice tuplă din cadrul unei relații/tabele are o identitate unică.

#### Constrângerea de integritate referențială

Această constrângere de integritate se aplică cheilor străine (foreing keys).

**Constrângerea de integritate referențială** presupune faptul că dacă există o cheie străină (foreign key) în cadrul unei relații/tabele, atunci valoarea acesteia trebuie să coincidă cu valoarea unei chei candidat (candidate key) din relația/tabela părinte sau valoarea acesteia trebuie să fie în totalitate null (toate atributele acesteia să fie null). Această constrângere asigură faptul că orice referire a unei tuple părinte este validă.

#### Constrângeri generale

**Constrângerile generale** reprezintă reguli de integritate specificate la nivelul bazei de date care definesc sau impun anumite constrângeri asupra datelor (exemplu: un student nu poate împrumuta mai mult de 5 cărți; un profesor nu poate predă mai mult de 20 de ore pe săptămână etc.).

În general, gradul de suport pentru specificarea constrângerilor generale variază în funcție de SGBD-ul ales.

## Bibliografie

- [1] Abraham Silberschat, Henry F. Korth, S. Sudarshan: **Database System Concepts (6th Edition)**, McGraw-Hill, ISBN 978-0-07-352332-3, 2011
- [2] Thomas M. Connolly, Carolyn E. Begg: **Database Systems: A Practical Approach to Design, Implementation, and Management (6th Edition)**, Pearson Education Limited, ISBN 10: 1-292-06118-9, ISBN 13: 978-1-292-06118-4, 2015
- [3] Carlos Coronel, Steven Morris: **Database Systems: Design, Implementation, and Management (11th Edition)**, ISBN-13: 978-1-285-19614-5, ISBN-10: 1-285-19614-7, Cengage Learning, 2015
- [4] Claudiu Botez, Cătălin Mironeanu, Doina Buzea: **Baze de date**, Politehnium, ISBN: 978-973-621-162-1, 2009

## Algebra relațională și calculul relațional

Algebra relațională și calculul relațional formează baza limbajelor relaționale.

**Algebra relațională** poate fi descrisă ca un limbaj procedural de nivel înalt: poate fi folosită pentru a specifica SGBDR-ului (RDBMS-ului) cum să construiească noi relații în baza relațiilor existente în baza de date.

**Calculul relațional** poate fi descris ca un limbaj declarativ (non-procedural): poate fi folosit pentru a formula definiția unei relații în termenii relațiilor existente în baza de date.

*Formal, algebra relațională și calculul relațional sunt echivalente: pentru fiecare expresie din algebra relațională, există o expresie echivalentă în calculul relațional (și vice-versa).*

Algebra relațională și calculul relațional au fost folosite ca bază pentru limbaje de manipulare a datelor (Data Manipulation Languages – DMLs) de nivel înalt în contextul bazelor de date relaționale.

Algebra relațională și calculul relațional specifică operațiile de bază pe care orice DML trebuie să le implementeze și oferă o bază de comparație cu alte limbaje relaționale.

Un **limbaj relațional complet** este un limbaj care poate genera orice relație care poate fi construită folosind calculul relațional.

### Algebra relațională

Datele din tabelele relaționale au o valoare limitată dacă nu pot fi manipulate pentru a genera informații utile. Algebra relațională descrie partea teoretică de manipulare a modelului relațional folosind operatori relaționali.

Gradul de conformare relațională a unui SGBDR (RDBMS) este dat de nivelul de suport al algebrei relaționale. Pentru a fi considerat relațional, un SGBD (DBMS) trebuie să implementeze un minim de operatori relaționali: selecție (select), proiecție (projection) și joncțiune (join).

Deoarece modelul relațional este axat pe principii matematice, manipularea datelor din baza de date poate fi descrisă în termeni matematici.

Totuși, utilizatorii și programatorii aplicațiilor de baze de date nu trebuie să folosească formule matematice pentru a manipula datele. În schimb, aceștia folosesc limbaje cum ar fi SQL care ascund partea matematică din spate.

Studiul algebrei relaționale este important deoarece, prin asimilarea principiilor acesteia, programatorii aplicațiilor de baze de date pot înțelege mai ușor tipurile de operații care pot fi efectuate asupra datelor și pot deprinde abilități pentru generarea unor interogări optimizate.

*Operațiile din algebra relațională lucrează cu una sau mai multe relații pentru a defini o nouă relație, fără a modifica relațiile originale.* Prin urmare, atât operanții cât și rezultatele operațiilor reprezintă relații și astfel, rezultatul unei operații poate fi folosit ca și operand în cadrul unei alte operații. Această proprietate poartă denumirea de **închidere (closure)**: se spune că relațiile sunt *închise* sub algebra relațională.

Algebra relațională este un limbaj de tipul *relation-at-a-time* în care toate tuplele, din una sau mai multe relații, sunt manipulate în aceeași instrucțiune.

În cadrul algebrei relaționale sunt 6 operații fundamentale:

- selecție (select)
- proiecție (projection)
- produs cartezian (cartesian product)
- reuniune (union)
- diferență (set difference)
- redenumire (rename)

Alte operații adiționale care pot fi derivate din operațiile fundamentale sunt:

- joncțiune (join)
- intersecție (set intersection)
- diviziune (division)

Alte operații care extind algebra relațională:

- complementare (complement)
- spargere (split)
- închidere tranzitivă (transitive closure)
- proiecție generalizată (generalized projection)
- etc.

ACESTE operații pot fi clasificate în:

- **operații unare**, care lucrează cu o singură relație: selecție, proiecție, complementare, spargere, închidere tranzitivă
- **operații binare**, care lucrează cu două relații: produs cartezian, reuniune, diferență, joncțiune, intersecție, diviziune

### Selectie (select)

Operația de selecție lucrează pe o singură relație  $r(R)$  și are ca rezultat o relație care conține doar tuplele din  $r(R)$  care satisfac o condiție specificată (predicat).

$$\sigma_{\text{predicat}}(r) = \{ t \mid t \in r \wedge \text{predicat}(t) = \text{true} \}$$

Exemplu:

- toți angajații care au salariu mai mare de 5000:  $\sigma_{\text{Salariu}>5000}(\text{Angajat})$
- toți studenții care sunt în anul 2 sau în anul 3:  $\sigma_{\text{An}=2 \vee \text{An}=3}(\text{Student})$

Predicate compuse pot fi generate folosind operatorii logici SI (AND -  $\wedge$ ), SAU (OR -  $\vee$ ), NEGARE (NOT -  $\neg$ ).

### Proiecție (projection)

Operația de proiecție lucrează pe o singură relație  $r(R)$  și are ca rezultat o relație care conține un subset vertical al lui  $r(R)$  ce conține valorile atributelor specificate și elimină duplicatele.

$$\pi_{A_1, A_2, \dots, A_k}(r),$$

unde  $A_1, A_2, \dots, A_k$  sunt numele atributelor după care se face proiecția.

Eliminarea duplicatelor este necesară, deoarece orice relație reprezintă un set.

ID	Nume	Prenume	Adresă	Data_nașterii	Salariu
1	Popescu	Bogdan	str. Palat nr. 8	12/04/1985	6500
2	Luca	Andrei	b-dul Socola nr. 16	11/09/1991	4000
3	Rusu	Ana	str. Conductelor nr. 7	23/11/1992	5200

Fig. 1. Relația Angajat

Exemplu:

- proiecție după ID, Nume, Prenume și Salariu din relația Angajat:  $\pi_{ID, Nume, Prenume, Salariu}(Angajat)$

ID	Nume	Prenume	Salariu
1	Popescu	Bogdan	6500
2	Luca	Andrei	4000
3	Rusu	Ana	5200

Fig. 2. Proiecție Angajat după atributele ID, Nume, Prenume și Salariu

Produs cartezian (cartesian product)

Operația de produs cartezian dintre două relații  $r(R)$  și  $s(S)$  are ca rezultat o relație care conține concatenarea fiecărei tuple din relația  $r(R)$  cu fiecare tuplă din relația  $s(S)$ . Prin urmare, relația rezultat va conține toate perechile posibile de tuple din cele două relații,  $r(R)$  și  $s(S)$ .

$$r \times s = \{ tu \mid t \in r \wedge u \in s \}$$

Dacă o relație are  $X$  tuple și  $N$  atrbute, iar celalătă relație are  $Y$  tuple și  $M$  atrbute, atunci operația de produs cartezian va genera o relație ce va conține  $X \times Y$  tuple și  $N+M$  atrbute. În cazul în care există atrbute cu același nume în cele două relații, acestea vor fi prefixate cu numele relațiilor pentru a păstra unicitatea numerelor atrbutelor în relația rezultat.

ID	Nume	Prenume
1	Popescu	Bogdan
2	Luca	Andrei

Fig. 3. Relația Persoană

ID_Hobby	Hobby
10	Schi
20	Patinaj
30	Citit

Fig. 4. Relația Hobby

Exemplu:

- produsul cartezian dintre relațiile Persoană și Hobby: Persoană x Hobby

ID	Nume	Prenume	ID_Hobby	Hobby
1	Popescu	Bogdan	10	Schi
1	Popescu	Bogdan	20	Patinaj
1	Popescu	Bogdan	30	Citit

2	Luca	Andrei	10	Schi
2	Luca	Andrei	20	Patinaj
2	Luca	Andrei	30	Citit

Fig. 5. Produs cartezian dintre relațiile Persoană și Hobby

### Reuniune (union)

Operația de reuniune a două relații  $r(R)$  și  $s(S)$  are ca rezultat o relație care conține toate tuplele din  $r(R)$  și  $s(S)$ , cu eliminarea dupliilor.

$$r \cup s = \{ t \mid t \in r \vee t \in s \}$$

*Eliminarea dupliilor este necesară, deoarece orice relație reprezintă un set.*

Dacă o relație are X tuple, iar celalătă relație are Y tuple, atunci operația de reuniune va genera o relație ce va conține cel mult  $X+Y$  tuple. Pentru a putea aplica operația de reuniune, schemele R și S ale celor două relații trebuie să fie compatibile:

- trebuie să aibă același număr de attribute
- attributele corespunzătoare din cele două relații trebuie să aibă același domeniu

ID	Nume	Prenume
1	Popescu	Bogdan
2	Luca	Andrei

Fig. 6. Relația Persoană\_1

ID	Nume	Prenume
1	Popescu	Bogdan
2	Apetrei	Dumitru
3	Dinică	Ioana
4	Dumitru	Georgiana

Fig. 7. Relația Persoană\_2

Exemplu:

- reuniunea dintre relațiile Persoană\_1 și Persoană\_2: Persoană\_1  $\cup$  Persoană\_2

ID	Nume	Prenume
1	Popescu	Bogdan
2	Luca	Andrei
2	Apetrei	Dumitru
3	Dinică	Ioana
4	Dumitru	Georgiana

Fig. 8. Reuniune dintre relațiile Persoană \_1 și Personă\_2

*Notă: Tupa <1, Popescu, Bogdan>, deși apare în fiecare relație sursă, apare doar o singură dată în relația rezultat. Acest lucru se întâmplă deoarece relația rezultat reprezintă un set, iar în cadrul unui set nu există duplicate.*

### Diferență (set difference)

Operația de diferență dintre două relații  $r(R)$  și  $s(S)$  are ca rezultat o relație care conține toate tuplele care sunt conținute în relația  $r(R)$ , dar nu sunt conținute în relația  $s(S)$ .

$$r - s = \{ t \mid t \in r \wedge t \notin s \}$$

Dacă relația  $r(R)$  are  $X$  tuple, iar relația  $s(S)$  are  $Y$  tuple, atunci operația de diferență va genera o relație ce va conține cel mult  $X$  tuple. Pentru a putea aplica operația de diferență, schemele  $R$  și  $S$  ale celor două relații trebuie să fie compatibile:

- trebuie să aibă același număr de atribute
- atributele corespunzătoare din cele două relații trebuie să aibă același domeniu

Exemplu:

- diferența dintre relațiile Persoană\_2 și Persoană\_1: Persoană\_2 – Persoană\_1

ID	Nume	Prenume
2	Apetrei	Dumitru
3	Dinică	Ioana
4	Dumitru	Georgiana

Fig. 9. Diferență dintre relațiile Persoană\_2 și Persoană\_1

### Redenumire (rename)

Operația de redenumire permite atribuirea unui nume pentru rezultatul unei expresii relaționale  $E$ . Odată atribuit, acest nume poate fi folosit în cadrul altor expresii relaționale. Acest operator poate fi folosit și pentru a schimba numele atributelor din rezultatul expresiei relaționale.

$$\rho_x(E)$$

Rezultatul operației de redenumire este expresia  $E$  sub numele  $x$ , păstrând numele atributelor.

$$\rho_{x(A_1, A_2, \dots A_k)}(E)$$

Rezultatul operației de redenumire este expresia  $E$  sub numele  $x$  și având numele atributelor  $A_1, A_2, \dots, A_k$ .

Exemplu:

- redenumire atribute Nume și Prenume din relația Persoană în Nume\_persoană și Prenume\_persoană, cu păstrarea numelui relației:  $\rho_{\text{Persoană}(ID, Nume\_persoană, Prenume\_persoană)}(\text{Persoană})$

ID	Nume_persoană	Prenume_persoană
1	Popescu	Bogdan
2	Luca	Andrei

Fig. 10. Redenumire atribute Nume și Prenume din relația Persoană în Nume\_persoană și Prenume\_persoană

## Joncțiune (join)

Operația de joncțiune dintre două relații  $r(R)$  și  $s(S)$  are ca rezultat o relație care conține concatenarea unor perechi de tuple din relațiile  $r(R)$  și  $s(S)$  ce satisfac o condiție specificată (predicat). Această operație poate fi privită ca fiind derivată din operația de produs cartezian dintre relațiile  $r(R)$  și  $s(S)$  peste care se aplică operația de selecție cu predicatul operației de joncțiune.

Există mai multe tipuri de joncțiuni:

- joncțiune theta sau joncțiune condițională (theta join, condition join)
- equijoncțiune (equijoin)
- joncțiune naturală (natural join)
- joncțiune externă (outer join)
- semijoncțiune (semijoin)

ID_a	Nume_a	Dept_ID	Salariu_a
1	A	10	1600
2	B	10	2100
3	C	20	3800
4	D	30	2400
5	E	40	3100
6	F	40	2500

Fig. 11. Relația Angajat

ID_m	Nume_m	Dept_ID	Salariu_m	Bonus
1	X	10	3000	20%
2	Y	20	6200	80%
3	Z	30	3500	50%
4	W	50	8000	100%

Fig. 12. Relația Manager

*Joncțiune theta (theta join) sau joncțiunea condițională (condition join)*

Joncțiunea theta dintre două relații  $r(R)$  și  $s(S)$  este reprezentată:

$$r \bowtie_{\text{predicat}} s$$

În cadrul unei joncțiuni theta, predicatul este compus din elemente de forma:

$$r.A_r \theta s.B_s$$

unde  $\theta$  este unul dintre operatorii de comparație  $<$ ,  $\leq$ ,  $>$ ,  $\geq$ ,  $=$ ,  $\neq$  ( $\neq$ ), iar  $A_r$  și  $B_s$  reprezintă atrbute din relațiile  $r(R)$ , respectiv  $s(S)$ .

Joncțiunea theta poate fi scrisă folosind operatorii de selecție și produs cartezian astfel:

$$r \bowtie_{\text{predicat}} s = \sigma_{\text{predicat}}(r \times s)$$

Gradul relației rezultat este egal cu suma gradelor relațiilor  $r(R)$  și  $s(S)$ .

Exemplu:

- joncțiunea theta care generează tuplele compuse din orice pereche angajat, manager în care angajatul câștigă mai mult decât managerul:  $\bowtie_{\text{Angajat.Salariu}_a > \text{Manager.Salariu}_m} \text{Manager}$

ID_a	Nume_a	Dept_ID	Salariu_a	ID_m	Nume_m	Dept_ID	Salariu_m	Bonus
3	C	20	3800	1	X	10	3000	20%
3	C	20	3800	3	Z	30	3500	50%
5	E	40	3100	1	X	10	3000	20%

Fig. 13. Joncțiune theta angajați care câștigă mai mult decât manageri

#### Equijoncțiune (equijoin)

Equijoncțiunea este un tip special de joncțiune theta în care toate elementele predicatului folosesc operatorul de comparație  $=$ . În acest caz este folosit termenul de equijoncțiune în loc de joncțiune theta.

Reprezentarea și gradul relației rezultat sunt aceleași ca și în cazul joncțiunii theta.

Exemplu:

- equijoncțiunea care generează tuplele compuse din orice pereche angajat cu manager-ul său (managerul departamentului angajatului):  $\bowtie_{\text{Angajat.Dept\_ID} = \text{Manager.Dept\_ID}} \text{Manager}$

ID_a	Nume_a	Dept_ID	Salariu_a	ID_m	Nume_m	Dept_ID	Salariu_m	Bonus
1	A	10	1600	1	X	10	3000	20%
2	B	10	2100	1	X	10	3000	20%
3	C	20	3800	2	Y	20	6200	80%
4	D	30	2400	3	Z	30	3500	50%

Fig. 14. Equijoncțiune angajați și managerii acestora

#### Joncțiune naturală (natural join)

Joncțiunea naturală dintre două relații  $r(R)$  și  $s(S)$  este reprezentată:

$$r \bowtie s$$

Joncțiunea naturală este o equijoncțiune peste toate atributele comune (atribute care au același nume) ( $R \cap S$ ). În rezultatul operației de joncțiune naturală, fiecare atribut comun va apărea *doar o singură dată*.

Gradul relației rezultat este egal cu suma gradelor relațiilor  $r(R)$  și  $s(S)$  minus numărul atributelor comune (deoarece acestea apar doar într-un singur exemplar în relația rezultat).

Exemplu:

- joncțiunea naturală care generează tuplele compuse din orice pereche angajat cu managerul său (managerul departamentului angajatului):  $\bowtie \text{Manager}$

ID_a	Nume_a	Dept_ID	Salariu_a	ID_m	Nume_m	Salariu_m	Bonus
1	A	10	1600	1	X	3000	20%
2	B	10	2100	1	X	3000	20%
3	C	20	3800	2	Y	6200	80%
4	D	30	2400	3	Z	3500	50%

Fig. 15. Joncțiune naturală angajați și managerii acestora

*Notă: Atributul comun este Dept\_ID și apare doar o singură dată în rezultate.*

### *Joncțiune externă (outer join)*

În cadrul unei joncțiuni dintre două relații  $r(R)$  și  $s(S)$ , nu există întotdeauna o corespondență între atributele joncțiunii pentru toate tuplele din aceste relații. Prin urmare, vor fi tuple care nu vor avea corespondent și care nu vor apărea în rezultatul joncțiunii. În cazul în care se dorește și generarea acestor tuple, se va folosi operatorul de joncțiune externă.

Există trei tipuri de joncțiune externă, reprezentate după cum urmează:

- joncțiune externă stânga (left outer join)
  - generează perechile de tuple corespondente
  - generează și tuplele din  $r(R)$  care nu au corespondent în  $s(S)$
  - toate tuplele din  $r(R)$  apar în rezultate, fie că au sau nu corespondent

$$r \bowtie_{\text{predicat}} s$$

Exemplu:

- joncțiunea externă stânga care generează tuplele compuse din orice pereche angajat cu manager-ul său (managerul departamentului angajatului):  $\text{Angajat} \bowtie_{\text{Angajat.Dept\_ID}=\text{Manager.Dept\_ID}} \text{Manager}$

ID_a	Nume_a	Dept_ID	Salariu_a	ID_m	Nume_m	Dept_ID	Salariu_m	Bonus
1	A	10	1600	1	X	10	3000	20%
2	B	10	2100	1	X	10	3000	20%
3	C	20	3800	2	Y	20	6200	80%
4	D	30	2400	3	Z	30	3500	50%
5	E	40	3100	NULL	NULL	NULL	NULL	NULL
6	F	40	2500	NULL	NULL	NULL	NULL	NULL

Fig. 16.Joncțiune externă stânga angajați și managerii acestora

- joncțiune externă dreapta (right outer join)
  - generează perechile de tuple corespondente
  - generează și tuplele din  $s(S)$  care nu au corespondent în  $r(R)$
  - toate tuplele din  $s(S)$  apar în rezultate, fie că au sau nu corespondent

$$r \bowtie_{\text{predicat}} s$$

Exemplu:

- joncțiunea externă dreapta care generează tuplele compuse din orice pereche angajat cu manager-ul său (managerul departamentului angajatului):  $\text{Angajat} \bowtie_{\text{Angajat.Dept\_ID}=\text{Manager.Dept\_ID}} \text{Manager}$

ID_a	Nume_a	Dept_ID	Salariu_a	ID_m	Nume_m	Dept_ID	Salariu_m	Bonus
1	A	10	1600	1	X	10	3000	20%
2	B	10	2100	1	X	10	3000	20%
3	C	20	3800	2	Y	20	6200	80%
4	D	30	2400	3	Z	30	3500	50%
NULL	NULL	NULL	NULL	4	W	50	8000	100%

Fig. 17.Joncțiune externă dreapta angajați și managerii acestora

- joncțiune externă totală (full outer join)
  - generează perechile de tuplele corespondente
  - generează și tuplele din r(R) care nu au corespondent în s(S)
  - generează și tuplele din s(S) care nu au corespondent în r(R)
  - toate tuplele din r(R) apar în rezultate, fie că au sau nu corespondent
  - toate tuplele din s(S) apar în rezultate, fie că au sau nu corespondent

$r \bowtie_{\text{predicat}} S$

Exemplu:

- joncțiunea externă totală care generează tuplele compuse din orice pereche angajat cu manager-ul său (managerul departamentului angajatului):  $\text{Angajat} \bowtie_{\text{Angajat.Dept\_ID} = \text{Manager.Dept\_ID}} \text{Manager}$

ID_a	Nume_a	Dept_ID	Salariu_a	ID_m	Nume_m	Dept_ID	Salariu_m	Bonus
1	A	10	1600	1	X	10	3000	20%
2	B	10	2100	1	X	10	3000	20%
3	C	20	3800	2	Y	20	6200	80%
4	D	30	2400	3	Z	30	3500	50%
5	E	40	3100	NULL	NULL	NULL	NULL	NULL
6	F	40	2500	NULL	NULL	NULL	NULL	NULL
NULL	NULL	NULL	NULL	4	W	50	8000	100%

Fig. 18.Joncțiune externă totală angajați și managerii acestora

*Dacă nu se specifică predicatul, joncțiunea devine joncțiune externă naturală.*

În cazul tuplelor care nu au tuple corespondente și totuși sunt generate conform tipului de joncțiune externă folosită, în perechea de tuple rezultată, tupla corespondent va conține null-uri pentru toate atributele sale.

Joncțiunea externă poate fi privită ca o extensie a celorlalte tipuri de joncțiuni. În acest context, o joncțiune externă poate fi văzută ca o combinație de joncțiune, reuniune, proiecție, diferență și produs cartezian. Astfel, rezultatul aplicării joncțiunii initiale (ce conține doar perechi de tuple corespondente) este reunit cu perechile de tuple care nu au corespondent și pentru care tupla corespondent a fost setată cu toate atributele pe null. Aceste perechi, în care o tuplă este setată pe null, sunt generate folosind produsul cartezian dintre tuplele care nu au corespondent și o tuplă cu atributele setate pe null. Tuplele care nu au corespondent pot fi generate folosind diferența dintre relația ce le conține și proiecția după atributele acesteia a relației rezultate din aplicarea joncțiunii initiale.

Joncțiune externă stânga (left outer join):

$$r \bowtie_{\text{predicat } S} = (r \bowtie_{\text{predicat } S}) \cup ((r - \pi_R(r \bowtie_{\text{predicat } S})) \times \{\langle \text{null}, \text{null}, \dots \rangle_{|S|}\})$$

sau

$$T_1 \leftarrow (r \bowtie_{\text{predicat } S})$$

$$T_2 \leftarrow r - \pi_R(T_1)$$

$$T \leftarrow T_1 \cup (T_2 \times \{\langle \text{null}, \text{null}, \dots \rangle_{|S|}\})$$

Joncțiune externă dreapta (right outer join):

$$r \bowtie_{\text{predicat } S} = (r \bowtie_{\text{predicat } S}) \cup (\{\langle \text{null}, \text{null}, \dots \rangle_{|R|}\} \times (s - \pi_S(r \bowtie_{\text{predicat } S})))$$

sau

$$T_1 \leftarrow (r \bowtie_{\text{predicat } S})$$

$$T_2 \leftarrow s - \pi_S(T_1)$$

$$T \leftarrow T_1 \cup (\{\langle \text{null}, \text{null}, \dots \rangle_{|R|}\} \times T_2)$$

Joncțiune externă totală (full outer join):

$$r \bowtie_{\text{predicat } S} = (r \bowtie_{\text{predicat } S}) \cup ((r - \pi_R(r \bowtie_{\text{predicat } S})) \times \{\langle \text{null}, \text{null}, \dots \rangle_{|S|}\}) \cup (\{\langle \text{null}, \text{null}, \dots \rangle_{|R|}\} \times (s - \pi_S(r \bowtie_{\text{predicat } S})))$$

sau

$$T_1 \leftarrow (r \bowtie_{\text{predicat } S})$$

$$T_2 \leftarrow r - \pi_R(T_1)$$

$$T_3 \leftarrow s - \pi_S(T_1)$$

$$T \leftarrow T_1 \cup (T_2 \times \{\langle \text{null}, \text{null}, \dots \rangle_{|S|}\}) \cup (\{\langle \text{null}, \text{null}, \dots \rangle_{|R|}\} \times T_3)$$

Joncțiune externă stânga naturală (left natural outer join):

$$r \bowtie S = \dots \text{ temă de gândire} \dots$$

Joncțiune externă dreapta naturală (right natural outer join):

$$r \bowtie S = \dots \text{ temă de gândire} \dots$$

Joncțiune externă totală naturală (full natural outer join):

$$r \bowtie S = \dots \text{ temă de gândire} \dots$$

### *Semijoncțiune (semijoin)*

Operația de semijoncțiune dintre două relații  $r(R)$  și  $s(S)$  are ca rezultat o relație care conține doar tuplele dintr-o singură relație ( $r(R)$  sau  $s(S)$ ) – în funcție de tipul de semijoncțiune), care au corespondență în contextul unei joncțiuni cu o condiție specificată (predicat). Această operație poate fi privită ca fiind derivată din operația de joncțiune dintre relațiile  $r(R)$  și  $s(S)$  peste care se aplică operația de proiecție după atributele din  $R$  sau  $S$ , în funcție de tipul de semijoncțiune.

Există două tipuri de semijoncțiune, reprezentate după cum urmează:

- semijoncțiune stânga (left semijoin)
  - generează tuplele din  $r(R)$  care au corespondență în  $r \bowtie_{\text{predicat}} s$

$$r \bowtie_{\text{predicat}} s$$

Exemplu:

- semijoncțiunea stânga care generează tuplele compuse din angajații pentru care există un manager (managerul departamentului angajatului):  $\text{Angajat} \bowtie_{\text{Angajat.Dept\_ID}=Manager.Dept\_ID} \text{Manager}$

ID_a	Nume_a	Dept_ID	Salariu_a
1	A	10	1600
2	B	10	2100
3	C	20	3800
4	D	30	2400

Fig. 19. Semijoncțiune stânga angajați și managerii acestora

- semijoncțiune dreapta (right semijoin)
  - generează tuplele din  $s(S)$  care au corespondență în  $r \bowtie_{\text{predicat}} s$

$$r \bowtie_{\text{predicat}} s$$

Exemplu:

- semijoncțiunea dreapta care generează tuplele compuse din managerii pentru care există angajați (în departamentul acestora):  $\text{Angajat} \bowtie_{\text{Angajat.Dept\_ID}=Manager.Dept\_ID} \text{Manager}$

ID_m	Nume_m	Dept_ID	Salariu_m	Bonus
1	X	10	3000	20%
2	Y	20	6200	80%
3	Z	30	3500	50%

Fig. 20. Semijoncțiune dreapta angajați și managerii acestora

*Dacă nu se specifică predicatul, semijoncțiunea devine semijoncțiune naturală.*

Semijoncțiune stânga (left semijoin):

$$r \bowtie_{\text{predicat}} s = \pi_R(r \bowtie_{\text{predicat}} s)$$

Semijoncțiune dreapta (right semijoin):

$$r \bowtie_{\text{predicat}} s = \pi_S(r \bowtie_{\text{predicat}} s)$$

Semijonctiune stânga naturală (left natural semijoin):

$$r \bowtie s = \pi_R(r \bowtie s)$$

Semijonctiune dreapta naturală (right natural semijoin):

$$r \bowtie s = \pi_S(r \bowtie s)$$

Intersecție (set intersection)

Operația de intersecție dintre două relații  $r(R)$  și  $s(S)$  are ca rezultat o relație care conține toate tuplele care se află atât în relația  $r(R)$  cât și în relația  $s(S)$ .

$$r \cap s = \{ t \mid t \in r \wedge t \in s \}$$

Dacă relația  $r(R)$  are  $X$  tuple, iar relația  $s(S)$  are  $Y$  tuple, atunci operația de intersecție va genera o relație ce va conține cel mult  $\min(X, Y)$  tuple. Pentru a putea aplica operația de intersecție, schemele  $R$  și  $S$  ale celor două relații trebuie să fie compatibile:

- trebuie să aibă același număr de atribute
- atributele corespunzătoare din cele două relații trebuie să aibă același domeniu

Derivare folosind operația de diferență:

$$r \cap s = r - (r - s)$$

ID	Limbaj
21	C
12	C++
33	Java
4	C#
50	Perl

Fig. 21. Relația Limbaje\_1

ID	Limbaj
17	SQL
33	Java
4	Python
12	C++
80	C#

Fig. 22. Relația Limbaje\_2

Exemplu:

- intersecția dintre relațiile Limbaje\_1 și Limbaje\_2: Limbaje\_1  $\cap$  Limbaje\_2

ID	Nume
12	C++
33	Java

Fig. 23. Intersecție dintre relațiile Limbaje\_1 și Limbaje\_2

### Diviziune (division)

Operația de diviziune dintre două relații  $r(R)$  și  $s(S)$ , are ca rezultat o relație care conține toate tuplele din  $r(R)$  proiectate peste setul de attribute  $R - S$  care, concatenate cu orice tuplă din  $s(S)$  se regăsesc în  $r(R)$ .

$$r \div s = \{ t \mid t \in \pi_{R-S}(r) \wedge (\forall u \in s)(tu \in r) \}$$

Dacă relația  $r(R)$  are  $X$  tuple, iar relația  $s(S)$  are  $Y$  tuple, atunci operația de diviziune va genera o relație ce va conține cel mult  $X/Y$  tuple. Pentru a putea aplica operația de diviziune,  $S$  trebuie să fie un subset a lui  $R$ :

$$S \subseteq R$$

Derivare folosind proiecție, produs cartezian și diferență:

$$r \div s = \pi_{R-S}(r) - \pi_{R-S}((\pi_{R-S}(r) \times s) - \pi_{R-S,S}(r))$$

sau

$$T_1 \leftarrow \pi_{R-S}(r)$$

$$T_2 \leftarrow \pi_{R-S}((T_1 \times s) - \pi_{R-S,S}(r))$$

$$T \leftarrow T_1 - T_2$$

ID_Student	Limbaj	Nivel
1	C	Începător
2	Java	Mediu
2	C++	Avansat
3	Java	Avansat
3	C#	Mediu
4	Java	Mediu
4	C++	Începător
5	Perl	Mediu
6	C++	Avansat
6	Java	Mediu
6	Python	Începător

Fig. 24. Relația Student2Limbaj

Limbaj	Nivel
C++	Avansat
Java	Mediu

Fig. 25. Relația Cerințe

Exemplu:

- diviziunea relației Student2Limbaj la relația Cerințe:  $\text{Student2Limbaj} \div \text{Cerințe}$

ID_Student
2
6

Fig. 26. Diviziune relație Student2Limbaj la relație Cerinte

Notă: Studentul cu ID-ul 4 nu apare în rezultate deoarece, chiar dacă are cunoștințe despre limbajele cerute (C++ și Java), nivelul acestora nu corespunde.

#### Complementare (complement)

Operația de complementare lucrează pe o singură relație  $r(R)$  și are ca rezultat o relație care conține toate tuplele generate de produsul cartezian al domeniilor asociate schemei  $R$  și care nu fac parte din relația  $r(R)$ .

Fie  $R = \{ A_1:D_1, A_2:D_2, \dots, A_n:D_n \}$ . Atunci:

$$\neg(r) = \{ t : \langle d_1, d_2, \dots, d_n \rangle \mid t.d_1 \in D_1 \wedge t.d_2 \in D_2 \wedge \dots \wedge t.d_n \in D_n \wedge t \notin r \}$$

#### Spargere (split)

Operația de spargere lucrează pe o singură relație  $r(R)$  și are ca rezultat două relații care conțin doar tuplele din  $r(R)$  care satisfac, respectiv nu satisfac, o condiție specificată (predicat).

Această operație reprezintă un caz special deoarece, spre deosebire de celelalte operații din algebra relațională, are ca rezultat două relații. Prin urmare, nu poate fi folosită în formarea expresiilor.

Suma numărului de tuple din relațiile rezultat este egală cu numărul de tuple din relația  $r(R)$ .

$$\text{SPLIT}_{\text{predicat}}(r) = (s, s'), \text{ unde}$$

$$s = \{ t \mid t \in r \wedge \text{predicat}(t) = \text{true} \}$$

$$s' = \{ t \mid t \in r \wedge \text{predicat}(t) = \text{false} \}$$

ID	Nume	Prenume	Medie
1	Popescu	Silviu	7.4
2	Panaite	Laura	8.5
3	Dumitrașcu	Nicoleta	9.6
4	Pop	Răzvan	5.7
5	Axinte	Iuliu	7.7

Fig. 27. Relația StudentInfo

Exemplu:

- spargerea relației StudentInfo în două relații cu studenții ce au medii peste, respectiv sub, nota 8.0:  $\text{SPLIT}_{\text{Medie}>=8.0}(\text{StudentInfo})$

ID	Nume	Prenume	Medie
2	Panaite	Laura	8.5
3	Dumitrașcu	Nicoleta	9.6

Fig. 28. Relația rezultat cu studenții ce au medii peste 8.0 (inclusiv)

ID	Nume	Prenume	Medie
1	Popescu	Silviu	7.4
4	Pop	Răzvan	5.7
5	Axinte	Iuliu	7.7

Fig. 29. Relația rezultat cu studenții ce au medii peste 8.0 (exclusiv)

#### Închidere tranzitivă (transitive closure)

Operația de închidere tranzitivă lucrează pe o singură relație  $r(R)$  și are ca rezultat o relație care conține toate tuplele care rezultă din închiderea tranzitivă a tuplelor din  $r(R)$ . Pentru a putea fi aplicată, relația  $r(R)$  trebuie să conțină două attribute  $A_1$  și  $A_2$  definite pe același domeniu  $D$ . Această operație se aplică succesiv până când toate tuplele închiderii tranzitive au fost generate.

Regula de generare a tuplelor:

$$\tau_0(r) = r$$

$$\begin{aligned} \tau_t(r) &= \tau_{t-1}(r) \cup \{ t : \langle d_x, d_z \rangle \mid (\exists t_1 : \langle d_x, d_y \rangle, t_2 : \langle d_y, d_z \rangle) (t_1 \in \tau_{t-1}(r) \wedge t_2 \in \tau_{t-1}(r) \wedge t_1.d_y \\ &= t_2.d_y \wedge t.d_x = t_1.d_x \wedge t.d_z = t_2.d_z \wedge t \notin \tau_{t-1}(r)) \} \end{aligned}$$

ID_Manager	ID_Angajat
1	3
2	4
2	5
2	6
3	7
3	8
8	9

Fig. 30. Relația Manager2Angajat

Exemplu:

- Închiderea tranzitivă a relației Manager2Angajat (managerii cu toți subalternii, indiferent la ce nivel se află aceștia în ierarhie)

ID_Manager	ID_Angajat	$\tau_0$	$\tau_1$	$\tau_2$
1	3	$\tau_0$	$\tau_1$	$\tau_2$
2	4	$\tau_0$	$\tau_1$	$\tau_2$
2	5	$\tau_0$	$\tau_1$	$\tau_2$
2	6	$\tau_0$	$\tau_1$	$\tau_2$
3	7	$\tau_0$	$\tau_1$	$\tau_2$
3	8	$\tau_0$	$\tau_1$	$\tau_2$
8	9	$\tau_0$	$\tau_1$	$\tau_2$
1	7	$\tau_1$	$\tau_2$	
1	8	$\tau_1$	$\tau_2$	
3	9	$\tau_1$	$\tau_2$	
1	9	$\tau_2$		

Fig. 31. Închidere tranzitivă relație Manager2Angajat

*Notă: Tupla <1,9> apare în a doua iterare a închiderii tranzitive, fiind generată de tuplele <1,3> și <3,9> sau de tuplele <1,8> și <8, 9>.*

### Proiecția generalizată (generalized projection)

Operația de proiecție generalizată extinde operație de proiecție oferind posibilitatea aplicării unor transformări (funcții) asupra atributelor relației sursă  $r(R)$ .

$$\pi_{F_1, F_2, \dots, F_k}(r),$$

unde  $F_1, F_2, \dots, F_k$  sunt expresii care pot conține constante și atribute ale relației  $r(R)$ . Rezultatul acestor transformări reprezintă valori alor unor noi atribute în relația rezultat. Aceste atribute nu au nume și trebuie denumite folosind fie operația de redenumire:

$$\rho_{x(A_1, A_2, \dots, A_k)}(\pi_{F_1, F_2, \dots, F_k}(r))$$

, sau o convenție de notație echivalentă:

$$\pi_{F_1 \text{ as } A_1, F_2 \text{ as } A_2, \dots, F_k \text{ as } A_k}(r)$$

### Operația de agregare

Operația de agregare aplică o listă specificată de funcții de agregare pe o relație  $r(R)$  și are ca rezultat o altă relație care conține valori aggregate ale tuplelor din relația originală.

Lista de funcții de agregare,  $FList$ , este de forma:  $\text{funcție\_agregat}_1(\text{atribut}_1)$ ,  $\text{funcție\_agregat}_2(\text{atribut}_2)$ , ...,  $\text{funcție\_agregat}_k(\text{atribut}_k)$ .

$$G_{FList}(r)$$

Principalele funcții de agregare sunt:

- COUNT – returnează numărul de valori ale atributului asociat
- SUM – returnează suma valorilor atributului asociat
- AVG – returnează media valorilor atributului asociat
- MIN – returnează valoarea minimă a atributului asociat
- MAX – returnează valoarea maximă a atributului asociat

Exemplu:

- calculul mediei tuturor studenților:  $G_{AVG(\text{Nota})}(\text{Student})$
- calculul numărului total de angajați, și al salariului maxim al acestora:  
 $G_{CNT(\text{AngajatID}), MAX(\text{Salariu})}(\text{Angajat})$

### Operația de grupare

Operația de grupare grupează tuplele relației  $r(R)$  după un set specificat de atribute de grupare și apoi aplică o listă specificată de funcții de agregare fiecărui grup. Relația rezultată are schema definită de atributele de grupare și conține câte o tuplă pentru fiecare grup format din relația originală. Aceste tuple conțin valori agregat aferente grupurilor.

Lista de atribute de grupare, AttrList, este de forma:  $A_1, A_2, \dots, A_q$ .

Lista de funcții de agregare, FList, este de forma: funcție\_agregat<sub>1</sub>(atribut<sub>1</sub>), funcție\_agregat<sub>2</sub>(atribut<sub>2</sub>), ..., funcție\_agregat<sub>k</sub>(atribut<sub>k</sub>).

$$\text{AttrList } G_{F\text{List}}(r)$$

Exemplu:

- calculul mediei pe grupe de studenți: Grupă  $G_{AVG(\text{Notă})}(\text{Student})$
- calculul salariului minim, mediu și maxim, în cadrul fiecărui departament, pentru fiecare tip de job: DepartamentID, JobID  $G_{MIN(\text{Salariu}), AVG(\text{Salariu}), MAX(\text{Salariu})}(\text{Angajat})$

### Operații de modificare a relațiilor existente

Fie  $r(R)$  o relație. Operațiile de modificare sunt:

- operația de ștergere: este obținută prin aplicarea operației de diferență și atribuirea rezultatului unor relații existente.

$$r \leftarrow r - E$$

unde  $E$  reprezintă o expresie din algebra relațională care determină tuplele ce urmează a fi șterse din relația  $r(R)$ .

- operația de inserare: este obținută prin aplicarea operației de reuniune și atribuirea rezultatului unor relații existente.

$$r \leftarrow r \cup E$$

unde  $E$  reprezintă o expresie din algebra relațională care determină tuplele ce urmează a fi inserate în relația  $r(R)$ .

- operația de actualizare: modifică una sau mai multe valori ale atributelor în cadrul tuplelor. Poate fi exprimată prin operații de ștergere și inserare. Operațiile de ștergere vor elimina tuplele vechi, iar operațiile de inserare vor adăuga noile tuple care conțin valorile atributelor modificate.

$$r \leftarrow r - E_{\text{vechi}}$$

$$r \leftarrow r \cup E_{\text{nou}}$$

## Manipularea "valorilor" NULL

"Valoarea" NULL reprezintă o valoare necunoscută sau inexistentă. Prin urmare, se stabilesc următoarele reguli:

- orice operație aritmetică care implică "valori" NULL este evaluată la NULL
- orice operație logică care implică "valori" NULL nu poate fi evaluată nici pe *true*, nici pe *false*. Prin urmare, o nouă "valoare booleană" este definită, și anume *unknown*.
- interacțiunea în cadrul operațiilor logice dintre *true*, *false* și *unknown* este:

$$\text{true} \wedge \text{unknown} = \text{unknown}$$

$$\text{false} \wedge \text{unknown} = \text{false}$$

$$\text{unknown} \wedge \text{unknown} = \text{unknown}$$

$$\text{true} \vee \text{unknown} = \text{true}$$

$$\text{false} \vee \text{unknown} = \text{unknown}$$

$$\text{unknown} \vee \text{unknown} = \text{unknown}$$

$$\neg \text{unknown} = \text{unknown}$$

Efectul "valorilor" NULL în cadrul operațiilor din algebra relațională:

- operația de selecție: când predicatul  $P$  din  $\sigma_P(r)$  este evaluat pe *unknown* pentru o tuplă, acea tuplă nu este inclusă în rezultate.
- operația de joncțiune: deoarece joncțiunile sunt definite în termenii operației de selecție peste produsul cartezian, operația de selecție este cea care determină includerea tuplelor în rezultate. Predicatul de joncțiune este predicatul operațiunii de selecție. Dacă valorile comparate pentru o pereche de tuple conțin și "valori" NULL, predicatul va fi evaluat pe *unknown*, și conform operației de selecție, perechea evaluată nu se va regăsi în rezultate.
- operația de proiecție: în algebra relațională, tuplele care nu conțin "valori" NULL și care au aceleași valori pentru atributele din pozițiile corespondente sunt considerate duplicate, și apar doar o singură dată în rezultate. Dacă există un subset din aceste atribută pentru care valorile sunt NULL în toate tuplele analizate, acestea sunt considerate duplicate și vor fi incluse doar o singură dată în rezultate. Exemplu: tupla  $\langle \text{val}_1, \text{val}_2, \dots, \text{val}_k, \text{NULL}_1, \text{NULL}_2, \dots, \text{NULL}_m \rangle$  este considerată tuplă dupătul a tuplei  $\langle \text{val}_1, \text{val}_2, \dots, \text{val}_k, \text{NULL}_1, \text{NULL}_2, \dots, \text{NULL}_m \rangle$ .
- operațiile de reuniune, intersecție și diferență: se aplică aceleași principii pentru eliminarea duplicatelor ca și în cazul operației de proiecție.
- operațiile de complementare: se aplică aceleași principii pentru duplicate ca și în cazul operației de proiecție.
- operațiile de spargere și închidere tranzitivă: se aplică aceleași principii de evaluare a predicatului ca și în cazul selecției.
- operația de proiecție generalizată: se aplică principiile de evaluare a expresiilor definite mai sus și principiile de eliminare a duplicatelor ca și în cazul operației de proiecție.
- operațiile de agregare: în formarea grupurilor, tuplele care au valori comune (inclusiv "valori" NULL) ale atributelor de grupare vor fi plasate în același grup. La aplicarea funcțiilor de grup, "valorile" NULL sunt ignorate.

## Calculul relațional

În cadrul algebrei relaționale există o anumită ordine explicit specificată care implică o strategie de evaluare a expresiilor. Acest lucru nu se întâmplă în cadrul calculului relațional.

Calculul relațional specifică **ce** date trebuie returnate și nu **cum** sunt acestea preluate. Acest lucru reprezintă principala deosebire dintre algebra relațională și calculul relațional.

Calculul relațional este bazat pe calculul predicatelor. Acesta are două forme:

- calculul relațional orientat pe tuplu (propus de Codd în anul 1972)
- calculul relațional orientat pe domeniu (propus de Lacroix and Pirotte în anul 1977)

O **expresie a calculului relațional** creează o nouă relație care este specificată folosind variabile tuplu (în cazul calculului relațional orientat pe tuplu) sau variabile domeniu (în cazul calculului relațional orientat pe domeniu). În cadrul unei astfel de expresii, nu se specifică nici o ordine a operațiilor pentru preluarea datelor. O expresie a calculului relațional specifică doar ce date trebuie returnate.

În cadrul calculului predicatelor, un **predicat** reprezintă o funcție booleană care acceptă unul sau mai multe argumente. Când aceste argumente primesc valori efective, funcția devine o **propoziție**, și este evaluată la valoarea adevărat (true) sau fals (false).

Dacă P este un predicat, setul tuturor obiectelor x pentru care P este adevărat este scris ca:

$$\{ x \mid P(x) \}$$

Predicatelor pot fi combinate în predicate compuse prin folosirea operatorilor logici **ȘI** (AND -  $\wedge$ ), **SAU** (OR -  $\vee$ ), **NEGARE** (NOT -  $\neg$ ).

Predicatelor pot fi clasificate după numărul de variabile astfel:

- predicate unare (cu o variabilă)
- predicate binare (cu două variabile)
- predicate n-are (cu n variabile)

În calculul relațional se folosesc următorii operatori:

- Operatori logici:
  - conjuncția - **ȘI** (AND -  $\wedge$ )
  - disjuncția - **SAU** (OR -  $\vee$ )
  - negația - **NEGARE** (NOT -  $\neg$ )
- Quantificatori:
  - cuantificatorul existențial ( $\exists$ )
  - cuantificatorul universal ( $\forall$ )

O variabilă tuplu (în cazul calculului relațional orientat pe tuplu) / domeniu (în cazul calculului relațional orientat pe domeniu) este **legată (bound)** dacă este cuantificată folosind cuantificatorul existențial sau universal. În caz contrar, variabila este **liberă (free)**.

*Utilizând operatorii calculului relațional se pot obține noi formule în baza celor existente.*

Într-o formulă, prioritatea operatorilor, în ordine descrescătoare, este:

- operatorii de comparație
- cuantificatorii (existențial -  $\exists$  și universal -  $\forall$ )
- negația - NEGARE (NOT -  $\neg$ )
- conjuncția - ȘI (AND -  $\wedge$ )
- disjuncția - SAU (OR -  $\vee$ )

Pentru a schimba ordinea de aplicare a operatorilor se pot folosi paranteze. Evaluarea formulelor se face de la stânga la dreapta, în ordinea priorității operatorilor și ținând cont de eventualele paranteze.

### Calculul relațional orientat pe tuplu

Calculul relațional orientat pe tuplu folosește variabile tuplu. O **variabilă tuplu** este o variabilă care poate reprezenta pe rând tuplele unei relații. Pentru a specifica că o variabilă tuplu  $t$  reprezintă tuplele relației r scriem:

$$t \in r \text{ sau } r(t)$$

O expresie din cadrul calculului relațional orientat pe tuplu are forma:

$$\{ t \mid F(t) \}$$

unde  $F$  este o **formulă bine formată (well-formed formula sau wff)**. Expresia de mai sus returnează toate tuplele  $t$  pentru care  $F(t)$  este adevărată.

Forma generalizată a unei expresii în cadrul calculului relațional orientat pe tuplu este:

$$\{ t_1, t_2, \dots, t_n \mid F(t_1, t_2, \dots, t_m) \}, n \leq m$$

sau

$$\{ t_1.A_{11}, t_1.A_{12}, \dots, t_1.A_{1k_1}, t_2.A_{21}, t_2.A_{22}, \dots, t_2.A_{2k_2}, t_n.A_{n1}, t_n.A_{n2}, \dots, t_n.A_{nk_n} \mid F(t_1, t_2, \dots, t_m) \}, n \leq m$$

unde  $t_1, t_2, \dots, t_n$  sunt variabile tuplu, iar  $A_{xy}$  sunt atributele relațiilor aferente variabilelor tuplu.

**Singurele variabile libere din cadrul unei expresii sunt cele care apar înaintea |. Simbolul | se citește "pentru care".**

La fel ca și în cazul algebrei relaționale, expresiile calculului relațional reprezintă relații.

Expresiile de mai sus generează o nouă relație,  $result$ , care conține toate tuplele care satisfac formula  $F$ . Atributele relației  $result$  sunt:

- fie definite direct în  $F$  (folosind operatori de comparație unde unul dintre operanzi este de forma  $t.\text{nume\_atribut}$ )
- fie sunt moștenite de la relația de bază (folosind un predicat de forma:  $t \in r$ )

Exemplu:

- toți angajații al căror salar este mai mare de 5000:

$$\{ a \mid a \in \text{Angajat} \wedge a.\text{Salariu} > 5000 \}$$

sau

$$\{ t \mid (\exists a)(a \in \text{Angajat} \wedge a.\text{Salariu} > 5000 \wedge t = a) \}$$

echivalentul în algebra relațională:  $\sigma_{\text{Salariu}>5000}(\text{Angajat})$

- numele și prenumele tuturor angajaților al căror salar este mai mare de 5000:

$$\{ a.\text{Nume}, a.\text{Prenume} \mid a \in \text{Angajat} \wedge a.\text{Salariu} > 5000 \}$$

sau

$$\{ t \mid (\exists a)(a \in \text{Angajat} \wedge a.\text{Salariu} > 5000 \wedge t.\text{Nume} = a.\text{Nume} \wedge t.\text{Prenume} = a.\text{Prenume}) \}$$

echivalentul în algebra relațională:  $\pi_{\text{Nume}, \text{Prenume}}(\sigma_{\text{Salariu}>5000}(\text{Angajat}))$

### Atomi și formule

Un **atom** reprezintă cea mai simplă construcție. Acesta mai poartă denumirea și de **formulă atomică**. Orice atom este o **formulă** și poate fi construit din următoarele elemente:

- nume de relații
- variabile tuplu
- constante
- operatori de comparație care folosesc ca operanți variabile și constante

Elementele unui atom poartă denumirea de **termeni**.

Tipuri de atomi:

- $t \in r$  sau  $r(t)$ 
  - $t$  este o variabilă tuplu din relația  $r$
- $t_x.A_x \theta t_y.A_y$ 
  - unde  $t_x$  și  $t_y$  sunt variabile tuplu,  $\theta$  este unul dintre operatorii de comparație  $<$ ,  $<=$ ,  $>$ ,  $>=$ ,  $=$ ,  $!=$  ( $<>$ ), iar  $A_x$  și  $A_y$  reprezintă atrbute cu domenii compatibile
- $t.A \theta k$ 
  - unde  $t$  este o variabilă tuplu,  $\theta$  este unul dintre operatorii de comparație  $<$ ,  $<=$ ,  $>$ ,  $>=$ ,  $=$ ,  $!=$  ( $<>$ ),  $A$  reprezintă un atrbut din variabila tuplu  $t$ , iar  $k$  reprezintă o constantă compatibilă cu domeniul atrbutului  $A$

Formulele se construiesc recursiv folosind următoarele reguli:

- un atom este o formulă
- dacă  $F_1$  și  $F_2$  sunt formule, atunci  $F_1 \wedge F_2$  și  $F_1 \vee F_2$ , sunt tot formule
- dacă  $F$  este o formulă, atunci  $\neg F$  este tot o formulă
- dacă  $F$  este o formulă cu variabila tuplu liberă  $t$ , atunci  $(\exists t)(F(t))$  și  $(\forall t)(F(t))$  sunt tot formule

### Cuantificatorii existențial ( $\exists$ ) și universal ( $\forall$ )

În cadrul expresiilor pot fi folosiți cuantificatorii existențial ( $\exists$ ) și universal ( $\forall$ ) pentru a specifica pe câte instanțe operează o anumită formulă.

În cazul folosirii cuantificatorului existențial ( $\exists$ ), formula  $(\exists t)(F(t))$  este adevărată dacă formula  $F$  cu variabila liberă  $t$  este adevărată pentru cel puțin o tuplă din universul tuplelor candidat. În caz contrar formula este falsă.

În cazul folosirii cuantificatorului universal ( $\forall$ ), formula  $(\forall t)(F(t))$  este adevărată dacă formula  $F$  cu variabila liberă  $t$  este adevărată pentru toate tuplele din universul tuplelor candidat. În caz contrar formula este falsă.

Reguli logice:

- implicare logică:  $(p \Rightarrow q)$  este echivalent cu  $(\neg p \vee q)$
- $(\forall x \in S)(F(x))$  este echivalent cu  $(\forall x)((x \in S) \Rightarrow F(x))$
- $(\exists x \in S)(F(x))$  este echivalent cu  $(\exists x)((x \in S) \wedge F(x))$

Legile lui De Morgan:

- $\neg(F_1(x) \wedge F_2(x))$  este echivalent cu  $\neg F_1(x) \vee \neg F_2(x)$
- $\neg(F_1(x) \vee F_2(x))$  este echivalent cu  $\neg F_1(x) \wedge \neg F_2(x)$

Dubla negație:

- $(\forall x)(F(x))$  este echivalent cu  $\neg(\exists x)(\neg F(x))$ 
  - "orice adult a fost copil"  $\Leftrightarrow$  "nu există adult care nu a fost copil"
- $(\exists x)(F(x))$  este echivalent cu  $\neg(\forall x)(\neg F(x))$ 
  - "există mașini electrice"  $\Leftrightarrow$  "nu orice mașină nu este electrică"

Prin generalizarea legilor lui De Morgan obținem:

- $(\forall x)(F(x))$  este echivalent cu  $\neg(\exists x)(\neg F(x))$ 
  - $(\forall x)(F_1(x) \wedge F_2(x))$  este echivalent cu  $\neg(\exists x)(\neg(F_1(x) \wedge F_2(x)))$  care este echivalent cu  $\neg(\exists x)(\neg F_1(x) \vee \neg F_2(x))$
  - $(\forall x)(F_1(x) \vee F_2(x))$  este echivalent cu  $\neg(\exists x)(\neg(F_1(x) \vee F_2(x)))$  care este echivalent cu  $\neg(\exists x)(\neg F_1(x) \wedge \neg F_2(x))$
- $(\exists x)(F(x))$  este echivalent cu  $\neg(\forall x)(\neg F(x))$ 
  - $(\exists x)(F_1(x) \wedge F_2(x))$  este echivalent cu  $\neg(\forall x)(\neg(F_1(x) \wedge F_2(x)))$  care este echivalent cu  $\neg(\forall x)(\neg F_1(x) \vee \neg F_2(x))$
  - $(\exists x)(F_1(x) \vee F_2(x))$  este echivalent cu  $\neg(\forall x)(\neg(F_1(x) \vee F_2(x)))$  care este echivalent cu  $\neg(\forall x)(\neg F_1(x) \wedge \neg F_2(x))$

Exemplu:

Presupunem că:

- relația Angajat are attributele: ID, Nume, Prenume, Salariu, Dept\_ID
- relația Departament are attributele ID\_D, Nume\_D

- toți angajații  
 $\{ a \mid a \in \text{Angajat} \}$
- angajații care au salariul 5000:  
 $\{ a \mid a \in \text{Angajat} \wedge a.\text{Salariu} = 5000 \}$
- toți angajații care lucrează în departamentul marketing:  
 $\{ a \mid a \in \text{Angajat} \wedge (\exists d)(d \in \text{Departament} \wedge a.\text{Dept_ID} = d.\text{ID_D} \wedge d.\text{Nume_D} = "Marketing") \}$
- toți angajații care au cel mai mare salarior în departamentul lor:  
 $\{ a_1 \mid a_1 \in \text{Angajat} \wedge \neg(\exists a_2)(a_2 \in \text{Angajat} \wedge a_1.\text{Dept_ID} = a_2.\text{Dept_ID} \wedge a_2.\text{Salariu} > a_1.\text{Salariu}) \}$   
 sau  
 $\{ a_1 \mid a_1 \in \text{Angajat} \wedge (\forall a_2)(a_2 \in \text{Angajat} \Rightarrow (a_1.\text{Dept_ID} \neq a_2.\text{Dept_ID} \vee a_2.\text{Salariu} \leq a_1.\text{Salariu})) \}$   
 sau  
 $\{ a_1 \mid a_1 \in \text{Angajat} \wedge (\forall a_2)(a_2 \notin \text{Angajat} \vee a_1.\text{Dept_ID} \neq a_2.\text{Dept_ID} \vee a_2.\text{Salariu} \leq a_1.\text{Salariu}) \}$
- toate perechile de nume de angajați care lucrează în același departament:  
 $\{ a_1.\text{Nume}, a_1.\text{Prenume}, a_2.\text{Nume}, a_2.\text{Prenume} \mid a_1 \in \text{Angajat} \wedge a_2 \in \text{Angajat} \wedge a_1.\text{Dept_ID} = a_2.\text{Dept_ID} \wedge a_1.\text{ID} \neq a_2.\text{ID} \}$   
 sau  
 $\{ t \mid (\exists a_1)(\exists a_2)(a_1 \in \text{Angajat} \wedge a_2 \in \text{Angajat} \wedge a_1.\text{Dept_ID} = a_2.\text{Dept_ID} \wedge a_1.\text{ID} \neq a_2.\text{ID} \wedge t.\text{Nume}_1 = a_1.\text{Nume} \wedge t.\text{Prenume}_1 = a_1.\text{Prenume} \wedge t.\text{Nume}_2 = a_2.\text{Nume} \wedge t.\text{Prenume}_2 = a_2.\text{Prenume}) \}$

### Calculul relațional orientat pe domeniu

Calculul relațional orientat pe domeniu folosește variabile domeniu. O **variabilă domeniu** este o variabilă care poate lua valori din domeniile atributelor.

O expresie din cadrul calculului relațional orientat pe domeniu are forma:

$$\{ \langle d_1, d_2, \dots, d_n \rangle \mid F(d_1, d_2, \dots, d_m) \}, n \leq m$$

unde  $d_x$  sunt variabile domeniu, iar  $F$  este o formulă bine formată.

**Singurele variabile libere din cadrul unei expresii sunt cele care apar înaintea / . Simbolul / se citește "pentru care".**

### Atomi și formule

Un **atom** reprezintă cea mai simplă construcție. Acesta mai poartă denumirea și de **formulă atomică**. Orice atom este o **formulă** și poate fi construit din următoarele elemente:

- nume de relații
- variabile domeniu
- constante
- operatori de comparație care folosesc ca operanți variabile și constante

Elementele unui atom poartă denumirea de **termeni**.

Tipuri de atomi:

- $\langle d_1, d_2, \dots, d_n \rangle \in r$  sau  $r(d_1, d_2, \dots, d_n)$

- $d_1, d_2, \dots, d_n$  sunt variabile domeniu ce reprezintă tuple din relația  $r$
- $d_x \theta d_y$ 
  - unde  $d_x$  și  $d_y$  sunt variabile domeniu cu domenii compatibile, iar  $\theta$  este unul dintre operatorii de comparație  $<$ ,  $\leq$ ,  $>$ ,  $\geq$ ,  $=$ ,  $\neq$  ( $\neq$ )
- $d \theta k$ 
  - unde  $d$  este o variabilă domeniu,  $\theta$  este unul dintre operatorii de comparație  $<$ ,  $\leq$ ,  $>$ ,  $\geq$ ,  $=$ ,  $\neq$  ( $\neq$ ), iar  $k$  reprezintă o constantă compatibilă cu domeniul variabilei domeniu  $d$

Formulele se construiesc recursiv folosind următoarele reguli:

- un atom este o formulă
- dacă  $F_1$  și  $F_2$  sunt formule, atunci  $F_1 \wedge F_2$  și  $F_1 \vee F_2$ , sunt tot formule
- dacă  $F$  este o formulă, atunci  $\neg F$  este tot o formulă
- dacă  $F$  este o formulă cu variabila domeniu liberă  $d$ , atunci  $(\exists d)(F(d))$  și  $(\forall d)(F(d))$  sunt tot formule

### Expresii sigure

Este posibil ca o expresie a calculului relational (orientat pe tuplu sau pe domeniu) să genereze un set infinit de rezultate.

Exemplu:

$$E = \{ t \mid t \notin r \}$$

$$E = \{ \langle d_1, d_2, \dots, d_n \rangle \mid \langle d_1, d_2, \dots, d_n \rangle \notin r \}$$

sau echivalentul

$$E = \{ t \mid \neg(t \in r) \}$$

$$E = \{ \langle d_1, d_2, \dots, d_n \rangle \mid \neg(\langle d_1, d_2, \dots, d_n \rangle \in r) \}$$

O astfel de expresie se spune că este **nesigură**. Pentru a evita acest lucru trebuie adăugată o restricție conform căreia toate valorile care apar în rezultate trebuie să aparțină domeniului expresiei  $E$ , notat cu  $\text{dom}(E)$ . Domeniul expresiei  $E$  este format din toate valorile:

- care apar explicit în expresia  $E$
- care apar în relațiile referite în expresia  $E$

Exemplu:

$$E1 = \{ t \mid t \in \text{Angajat} \wedge t.\text{Salariu} > 5000 \wedge t.\text{Salariu} < 10000 \}$$

$$E1 = \{ \langle ID, Nume, Prenume, Adresă, Data_nășterii, Salariu \rangle \mid \langle ID, Nume, Prenume, Adresă, Data_nășterii, Salariu \rangle \in \text{Angajat} \wedge \text{Salariu} > 5000 \wedge \text{Salariu} < 10000 \}$$

Domeniul expresiei este reprezentat de reuniunea tuturor domeniilor atributelor relației Angajat la care se adaugă valorile referite direct în expresie: 5000, respectiv 10000.

$$E2 = \{ t \mid t \in \text{Angajat} \}$$

$E2 = \{ \langle ID, Nume, Prenume, Adresă, Data_nașterii, Salariu \rangle \mid \langle ID, Nume, Prenume, Adresă, Data_nașterii, Salariu \rangle \in \text{Angajat} \}$

$E3 = \{ t \mid t \notin \text{Angajat} \}$

$E3 = \{ \langle ID, Nume, Prenume, Adresă, Data_nașterii, Salariu \rangle \mid \langle ID, Nume, Prenume, Adresă, Data_nașterii, Salariu \rangle \notin \text{Angajat} \}$

Domeniul expresiei E2 **este același** cu domeniul expresiei E3, și anume: reuniunea tuturor domeniilor atributelor relației Angajat.

O expresie E se numește **sigură** dacă toate valorile care apar în rezultate aparțin domeniului acesteia,  $\text{dom}(E)$ . O expresie sigură returnează întotdeauna un număr finit de rezultate.

Exemplu:

Presupunem că:

- relația Angajat are attributele: ID, Nume, Prenume, Salariu, Dept\_ID
- relația Departament are attributele: ID\_D, Nume\_D

*Numele variabilelor tuplu pot fi diferite de numele atributelor pe care le reprezintă.*

- numele tuturor angajaților:  
 $\{ \langle \text{Nume}, \text{Prenume} \rangle \mid (\exists \text{ID}, \text{Salariu}, \text{Dept\_ID})(\langle \text{ID}, \text{Nume}, \text{Prenume}, \text{Salariu}, \text{Dept\_ID} \rangle \in \text{Angajat}) \}$
- numele angajaților care au salariul 5000:  
 $\{ \langle \text{Nume}, \text{Prenume} \rangle \mid (\exists \text{ID}, \text{Dept\_ID})(\langle \text{ID}, \text{Nume}, \text{Prenume}, 5000, \text{Dept\_ID} \rangle \in \text{Angajat}) \}$
- numele angajaților care lucrează în departamentul marketing:  
 $\{ \langle \text{Nume}, \text{Prenume} \rangle \mid (\exists \text{ID}, \text{Salariu}, \text{Dept\_ID}, \text{Nume\_D})(\langle \text{ID}, \text{Nume}, \text{Prenume}, \text{Salariu}, \text{Dept\_ID} \rangle \in \text{Angajat} \wedge \langle \text{Dept\_ID}, \text{Nume\_D} \rangle \in \text{Departament} \wedge \text{Nume\_D} = \text{"Marketing"}) \}$
- numele angajaților care au cel mai mare salariu în departamentul lor:  
 $\{ \langle \text{Nume}_1, \text{Prenume}_1 \rangle \mid (\exists \text{ID}_1, \text{Salariu}_1, \text{Dept\_ID}_1)(\neg(\exists \text{ID}_2, \text{Nume}_2, \text{Prenume}_2, \text{Salariu}_2)(\langle \text{ID}_1, \text{Nume}_1, \text{Prenume}_1, \text{Salariu}_1, \text{Dept\_ID}_1 \rangle \in \text{Angajat} \wedge \langle \text{ID}_2, \text{Nume}, \text{Prenume}, \text{Salariu}_2, \text{Dept\_ID} \rangle \in \text{Angajat} \wedge \text{Salariu}_2 > \text{Salariu}_1)) \}$   
 sau  
 $\{ \langle \text{Nume}_1, \text{Prenume}_1 \rangle \mid (\exists \text{ID}_1, \text{Salariu}_1, \text{Dept\_ID}_1)(\forall \text{ID}_2, \text{Nume}_2, \text{Prenume}_2, \text{Salariu}_2, \text{DeptID}_2)(\langle \text{ID}_1, \text{Nume}_1, \text{Prenume}_1, \text{Salariu}_1, \text{Dept\_ID}_1 \rangle \in \text{Angajat} \wedge \langle \text{ID}_2, \text{Nume}, \text{Prenume}, \text{Salariu}_2, \text{Dept\_ID}_2 \rangle \in \text{Angajat} \Rightarrow (\text{Dept\_ID}_1 \neq \text{Dept\_ID}_2 \vee \text{Salariu}_2 \leq \text{Salariu}_1)) \}$   
 sau  
 $\{ \langle \text{Nume}_1, \text{Prenume}_1 \rangle \mid (\exists \text{ID}_1, \text{Salariu}_1, \text{Dept\_ID}_1)(\forall \text{ID}_2, \text{Nume}_2, \text{Prenume}_2, \text{Salariu}_2, \text{DeptID}_2)(\langle \text{ID}_1, \text{Nume}_1, \text{Prenume}_1, \text{Salariu}_1, \text{Dept\_ID}_1 \rangle \in \text{Angajat} \wedge \langle \text{ID}_2, \text{Nume}, \text{Prenume}, \text{Salariu}_2, \text{Dept\_ID}_2 \rangle \notin \text{Angajat} \vee \text{Dept\_ID}_2 \neq \text{Dept\_ID}_1 \vee \text{Salariu}_2 \leq \text{Salariu}_1) \}$
- toate perechile de ID-uri de angajați care lucrează în același departament:  
 $\{ \langle \text{ID}_1, \text{ID}_2 \rangle \mid (\exists \text{Nume}_1, \text{Prenume}_1, \text{Salariu}_1, \text{Dept\_ID}, \text{Nume}_2, \text{Prenume}_2, \text{Salariu}_2)(\langle \text{ID}_1, \text{Nume}_1, \text{Prenume}_1, \text{Salariu}_1, \text{Dept\_ID} \rangle \in \text{Angajat} \wedge \langle \text{ID}_2, \text{Nume}_2, \text{Prenume}_2, \text{Salariu}_2, \text{Dept\_ID} \rangle \in \text{Angajat} \wedge \text{ID}_1 \neq \text{ID}_2) \}$   
 sau

$$\{ \langle \text{ID\_A1}, \text{ID\_A2} \rangle \mid (\exists \text{ID}_1, \text{Nume}_1, \text{Prenume}_1, \text{Salariu}_1, \text{Dept\_ID}, \text{ID}_2, \text{Nume}_2, \text{Prenume}_2, \text{Salariu}_2) (\langle \text{ID}_1, \text{Nume}_1, \text{Prenume}_1, \text{Salariu}_1, \text{Dept\_ID} \rangle \in \text{Angajat} \wedge \langle \text{ID}_2, \text{Nume}_2, \text{Prenume}_2, \text{Salariu}_2, \text{Dept\_ID} \rangle \in \text{Angajat} \wedge \text{ID}_1 \neq \text{ID}_2 \wedge \text{ID\_A1} = \text{ID}_1 \wedge \text{ID\_A2} = \text{ID}_2) \}$$

Exemple folosind calculul relațional:

ID_Student	Nume	Prenume
1	Popescu	Silviu
2	Panaite	Laura
3	Dumitrașcu	Nicoleta
4	Pop	Răzvan
5	Axinte	Iuliu

Fig. 32. Relația Student

ID_Curs	Nume_curs
1	PA
2	BD
3	PPAD
4	FSD
5	SO

Fig. 33. Relația Curs

ID_S	ID_C	Notă
1	1	9
1	3	7.5
2	2	8
4	3	9.2
4	2	10

Fig. 34. Relația Înscriere

Numele tuturor studenților care s-au înscris la cursul cu ID-ul 3:

Calcul relațional orientat pe tuplu:

$$\{ s.\text{Nume}, s.\text{Prenume} \mid s \in \text{Student} \wedge (\exists i)(i \in \text{Înscriere} \wedge i.\text{ID\_S} = s.\text{ID\_Student} \wedge i.\text{ID\_C} = 3) \}$$

sau

$$\{ t \mid (\exists s)(\exists i)(s \in \text{Student} \wedge i \in \text{Înscriere} \wedge i.\text{ID\_S} = s.\text{ID\_Student} \wedge i.\text{ID\_C} = 3 \wedge t.\text{Nume} = s.\text{Nume} \wedge t.\text{Prenume} = s.\text{Prenume}) \}$$

Calcul relațional orientat pe domeniu:

$$\{ \langle \text{Nume}, \text{Prenume} \rangle \mid (\exists \text{ID\_Student}, \text{ID\_S}, \text{ID\_C}, \text{Notă})(\langle \text{ID\_Student}, \text{Nume}, \text{Prenume} \rangle \in \text{Student} \wedge \langle \text{ID\_S}, \text{ID\_C}, \text{Notă} \rangle \in \text{Înscriere} \wedge \text{ID\_Student} = \text{ID\_S} \wedge \text{ID\_C} = 3) \}$$

sau

$\{ \langle \text{Nume, Prenume} \rangle \mid (\exists \text{ID\_Student}, \text{ID\_C}, \text{Notă})(\langle \text{ID\_Student}, \text{Nume, Prenume} \rangle \in \text{Student} \wedge \langle \text{ID\_Student}, \text{ID\_C}, \text{Notă} \rangle \in \text{Înscriere} \wedge \text{ID\_C} = 3) \}$

sau

$\{ \langle \text{Nume, Prenume} \rangle \mid (\exists \text{ID\_Student}, \text{Notă})(\langle \text{ID\_Student}, \text{Nume, Prenume} \rangle \in \text{Student} \wedge \langle \text{ID\_Student}, 3, \text{Notă} \rangle \in \text{Înscriere}) \}$

sau

$\{ \langle \text{NumeS, PrenumeS} \rangle \mid (\exists \text{ID\_Student}, \text{Nume, Prenume}, \text{Notă})(\langle \text{ID\_Student}, \text{Nume, Prenume} \rangle \in \text{Student} \wedge \langle \text{ID\_Student}, 3, \text{Notă} \rangle \in \text{Înscriere} \wedge \text{NumeS} = \text{Nume} \wedge \text{PrenumeS} = \text{Prenume}) \}$

*Numele tuturor studentilor care s-au înscris la cursul BD:*

Calcul relațional orientat pe tuplu:

$\{ \text{s.Nume, s.Prenume} \mid \text{s} \in \text{Student} \wedge (\exists \text{c})(\exists \text{i})(\text{c} \in \text{Curs} \wedge \text{i} \in \text{Înscriere} \wedge \text{c.Nume_curs} = "BD" \wedge \text{c.ID_Curs} = \text{i.ID_C} \wedge \text{i.ID_S} = \text{s.ID_Student}) \}$

sau

$\{ \text{t} \mid (\exists \text{s})(\exists \text{c})(\exists \text{i})(\text{s} \in \text{Student} \wedge \text{c} \in \text{Curs} \wedge \text{i} \in \text{Înscriere} \wedge \text{c.Nume_curs} = "BD" \wedge \text{c.ID_Curs} = \text{i.ID_C} \wedge \text{i.ID_S} = \text{s.ID_Student} \wedge \text{t.Nume} = \text{s.Nume} \wedge \text{t.Prenume} = \text{s.Prenume}) \}$

Calcul relațional orientat pe domeniu:

$\{ \langle \text{Nume, Prenume} \rangle \mid (\exists \text{ID\_Student}, \text{ID_Curs}, \text{Notă})(\langle \text{ID\_Student}, \text{Nume, Prenume} \rangle \in \text{Student} \wedge \langle \text{ID_Curs}, "BD" \rangle \in \text{Curs} \wedge \langle \text{ID\_Student}, \text{ID_Curs}, \text{Notă} \rangle \in \text{Înscriere}) \}$

sau

$\{ \langle \text{NumeS, PrenumeS} \rangle \mid (\exists \text{ID\_Student}, \text{Nume, Prenume}, \text{ID_Curs}, \text{Notă})(\langle \text{ID\_Student}, \text{Nume, Prenume} \rangle \in \text{Student} \wedge \langle \text{ID_Curs}, "BD" \rangle \in \text{Curs} \wedge \langle \text{ID\_Student}, \text{ID_Curs}, \text{Notă} \rangle \in \text{Înscriere} \wedge \text{NumeS} = \text{Nume} \wedge \text{PrenumeS} = \text{Prenume}) \}$

*Diviziune (Fig. 24, Fig. 25): ID-urile tuturor studentilor din relația Student2Limbaj care satisfac cerințele din relația Cerințe:*

Calcul relațional orientat pe tuplu:

$\{ \text{s.ID_Student} \mid \text{s} \in \text{Student2Limbaj} \wedge (\forall \text{c} \in \text{Cerințe})(\langle \text{s.ID_Student}, \text{c.Limbaj}, \text{c.Nivel} \rangle \in \text{Student2Limbaj}) \}$

sau

$\{ \text{s.ID_Student} \mid \text{s} \in \text{Student2Limbaj} \wedge (\forall \text{c})(\langle \text{c} \in \text{Cerințe} \Rightarrow \langle \text{s.ID_Student}, \text{c.Limbaj}, \text{c.Nivel} \rangle \in \text{Student2Limbaj} \rangle) \}$

sau

$\{ \text{t} \mid (\exists \text{s})(\text{s} \in \text{Student2Limbaj} \wedge (\forall \text{c})(\langle \text{c} \in \text{Cerințe} \Rightarrow \langle \text{s.ID_Student}, \text{c.Limbaj}, \text{c.Nivel} \rangle \in \text{Student2Limbaj} \rangle) \wedge \text{t.ID_Student} = \text{s.ID_Student}) \}$

Calcul relațional orientat pe domeniu:

$$\{ \langle \text{ID\_Student} \rangle \mid (\exists \text{Limbaj}, \text{Nivel}) (\langle \text{ID\_Student}, \text{Limbaj}, \text{Nivel} \rangle \in \text{Student2Limbaj} \wedge (\forall \langle \text{LimbajC}, \text{NivelC} \rangle \in \text{Cerințe}) (\langle \text{ID\_Student}, \text{LimbajC}, \text{NivelC} \rangle \in \text{Student2Limbaj})) \}$$

sau

$$\{ \langle \text{ID\_Student} \rangle \mid (\exists \text{Limbaj}, \text{Nivel}) (\langle \text{ID\_Student}, \text{Limbaj}, \text{Nivel} \rangle \in \text{Student2Limbaj} \wedge (\forall \text{LimbajC}, \text{NivelC}) ((\langle \text{LimbajC}, \text{NivelC} \rangle \in \text{Cerințe}) \Rightarrow (\langle \text{ID\_Student}, \text{LimbajC}, \text{NivelC} \rangle \in \text{Student2Limbaj}))) \}$$

sau

$$\{ \langle \text{ID\_S} \rangle \mid (\exists \text{ID\_Student}, \text{Limbaj}, \text{Nivel}) (\langle \text{ID\_Student}, \text{Limbaj}, \text{Nivel} \rangle \in \text{Student2Limbaj} \wedge (\forall \text{LimbajC}, \text{NivelC}) ((\langle \text{LimbajC}, \text{NivelC} \rangle \in \text{Cerințe}) \Rightarrow (\langle \text{ID\_Student}, \text{LimbajC}, \text{NivelC} \rangle \in \text{Student2Limbaj}))) \wedge \text{ID\_S} = \text{ID\_Student} \}$$

## Bibliografie

- [1] Abraham Silberschat, Henry F. Korth, S. Sudarshan: ***Database System Concepts (6th Edition)***, McGraw-Hill, ISBN 978-0-07-352332-3, 2011
- [2] Thomas M. Connolly, Carolyn E. Begg: ***Database Systems: A Practical Approach to Design, Implementation, and Management (6th Edition)***, Pearson Education Limited, ISBN 10: 1-292-06118-9, ISBN 13: 978-1-292-06118-4, 2015
- [3] Carlos Coronel, Steven Morris: ***Database Systems: Design, Implementation, and Management (11th Edition)***, ISBN-13: 978-1-285-19614-5, ISBN-10: 1-285-19614-7, Cengage Learning, 2015
- [4] Claudiu Botez, Cătălin Mironeanu, Doina Buzea: ***Baze de date***, Politehnium, ISBN: 978-973-621-162-1, 2009

## Normalizarea bazelor de date

Reprezintă procesul de organizare (fără a pierde din informații) a atributelor și tabelelor dintr-o bază de date relațională, cu scopul de a minimiza redundanța datelor și implicit de a minimiza potențialele erori care pot apărea în manipularea datelor.

Limbajul pentru accesarea și manipularea datelor dintr-o bază de date relațională este SQL (Structured Query Language). SQL poate fi folosit pentru:

- inserare date noi
- ștergere date existente
- actualizare date existente

În cadrul unei baze de date relaționale ne-normalizate, pot apărea 3 anomalii în manipularea datelor:

- Anomalie la inserare – imposibilitatea inserării anumitor tipuri de date
- Anomalie la ștergere – ștergerea unor date poate declanșa ștergerea unor date adiționale care în mod normal ar fi trebuit menținute
- Anomalie la actualizare – apariția unor inconsistențe ale datelor

Edgar F. Codd, inventatorul modelului relațional, a introdus conceptul de normalizare.

Forme normale:

- Prima formă normală (1970, Edgar F. Codd)
- A doua formă normală (1971, Edgar F. Codd)
- A treia formă normală (1971, Edgar F. Codd)
- Forma normală Boyce-Codd (1974, Edgar F. Codd și Raymond F. Boyce)
- A patra formă normală (1977, Ronald Fagin)
- A cincea formă normală (1979, Ronald Fagin)

## Definiții

- Super cheie

O super cheie reprezintă un set de atribute pentru care nu există nici o pereche de tuple care să aibă aceleași valori pentru aceste atribute. Prin urmare, o super cheie identifică în mod unic fiecare tuplă.

- Super cheie minimală

O super cheie este minimală dacă nici un subset strict al acesteia nu este la rândul său super cheie. Cu alte cuvinte, o super cheie minimală reprezintă un set minim de atribute pentru care o tuplă poate fi identificată în mod unic.

- Cheie candidat

O cheie candidat este o super cheie minimală.

- Cheie primară

O tabelă poate avea o singură cheie primară. Aceasta poate fi oricare dintre cheile candidat.

- Atribut cheie

Un atribut cheie face parte dintr-o cheie candidat. Toate atributele unei chei candidat sunt atribute cheie.

- Atribut non-cheie

Un atribut non-cheie nu face parte din nici o cheie candidat.

## Prima formă normală

O relație este în prima formă normală dacă satisfac următoarele condiții:

- un atribut conține valori atomice din domeniul său (și nu grupuri de astfel de valori)
- nu conține grupuri care se repetă

**Tabela angajați**

<u>AngajatID</u>	Nume	Prenume	Telefon
101	Popescu	Ion	123-111-222
102	Dumitru	Aurel	123-333-444
103	Ilie	Roxana	123-555-666

Dacă la un moment dat apare nevoie de a avea mai multe numere de telefon pentru un angajat, un designer poate decide să rețină mai multe valori în câmpul Telefon ca în exemplul de mai jos.

**Tabela angajați**

<u>AngajatID</u>	Nume	Prenume	Telefon
101	Popescu	Ion	123-111-222
102	Dumitru	Aurel	123-333-444
			123-333-445
103	Ilie	Roxana	123-555-666

Această reprezentare nu este în prima formă normală deoarece câmpul Telefon conține mai multe valori din domeniul acestuia.

O altă modalitate ar fi să includă coloane repetitive pentru Telefon.

**Tabela angajați**

<u>AngajatID</u>	Nume	Prenume	Telefon_1	Telefon_2
101	Popescu	Ion	123-111-222	
102	Dumitru	Aurel	123-333-444	123-333-445
103	Ilie	Roxana	123-555-666	

Deși nu este cuprinsă în formularea originală dată de Codd, nici această reprezentare nu este considerată a fi în prima formă normală deoarece atributul Telefon se repetă (coloanele Telefon\_1 și Telefon\_2).

Pentru a fi în prima formă normală, tabela poate fi reprezentată astfel:

**Tabela angajați**

<u>AngajatID</u>	Nume	Prenume	Telefon
101	Popescu	Ion	123-111-222
102	Dumitru	Aurel	123-333-444
102	Dumitru	Aurel	123-333-445
103	Ilie	Roxana	123-555-666

Această reprezentare, deși este în prima formă normală, nu este în a doua formă normală.

O reprezentare care este și în a doua și a treia forma normală este dată mai jos:

**Tabela angajați**

<u>AngajatID</u>	Nume	Prenume
101	Popescu	Ion
102	Dumitru	Aurel
103	Ilie	Roxana

**Tabela telefoane\_angajați**

<u>AngajatID</u>	Telefon
101	123-111-222
102	123-333-444
102	123-333-445
103	123-555-666

## A doua formă normală

O relație este în a doua formă normală dacă satisface următoarele condiții:

- este în prima formă normală
- toate atributele non-cheie depind în totalitate de **TOATE** cheile candidat

Cu alte cuvinte, dacă un atribut non-cheie depinde doar parțial de o cheie candidat (deinde de un subset strict al acesteia), atunci relația nu se află în a doua formă normală.

Dacă cheile candidat nu sunt de tipul cheie multiplă (sunt chei simple care conțin un singur atribut), o relație care este în prima formă normală este automat și în a doua formă normală.

**Tabela cumpărături**

<u>ClientID</u>	<u>MagazinID</u>	Locație
1	10	Iași
1	30	Timișoara
2	20	București
3	30	Timișoara
4	20	București

Cheia primară a tabelei este <ClientID, MagazinID>. Atributul non-cheie Locație depinde doar de MagazinID, acesta din urmă făcând parte din cheia primară. Astfel, atributul non-cheie depinde doar parțial de cheia primară (care este și cheie candidat). Prin urmare, tabela nu este în a doua formă normală.

Pentru a aduce tabela în a doua formă normală, aceasta trebuie spartă în două tabele după cum urmează:

**Tabela cumpărături**

<u>ClientID</u>	<u>MagazinID</u>
1	10
1	30
2	20
3	30
4	20

**Tabela magazine**

<u>MagazinID</u>	<u>Locație</u>
10	Iași
20	București
30	Timișoara

Prin spargerea tabelei inițiale, s-a eliminat dependența parțială de cheia primară. În tabela rezultată magazine, atributul non-cheie Locație depinde acum în totalitate de cheia primară <MagazinID>.

### A treia formă normală

O relație este în a treia formă normală dacă satisfac următoarele condiții:

- este în a doua formă normală
- toate atributele non-cheie sunt direct (non-tranzitiv) dependente de **TOATE** cheile candidat

**Tabela filme**

<u>FilmID</u>	<u>GenID</u>	<u>Gen</u>	<u>An</u>
1	10	S.F.	2002
2	20	Horror	2014
3	10	S.F.	1996
4	20	Horror	2008
5	30	Acțiune	2007

Tabela filme nu este în a treia formă normală deoarece atributul Gen depinde tranzitiv de cheia primară (care este și cheie candidat) <FilmID> prin intermediul atributului GenID. Altfel spus: FilmID -> GenID -> Gen.

Pentru a aduce tabela în a treia formă normală, aceasta trebuie spartă în două tabele după cum urmează:

**Tabela filme**

FilmID	GenID	An
1	10	2002
2	20	2014
3	10	1996
4	20	2008
5	30	2007

**Tabela genuri**

GenID	Gen
10	S.F.
20	Horror
30	Acțiune

După spargerea tabelei inițiale, toate atributele non-cheie din cele două tabele rezultante sunt direct dependente funcțional de cheia primară. În tabela filme:  $\langle \text{FilmID} \rangle \rightarrow \text{GenID}$ ,  $\langle \text{FilmID} \rangle \rightarrow \text{An}$ . În tabela genuri:  $\langle \text{GenID} \rangle \rightarrow \text{Gen}$ .

### Forma normală Boyce-Codd

Reprezintă o versiune mai puternică a celei de-a treia forme normale.

O relație este în forma normală Boyce-Codd dacă și numai dacă pentru fiecare dependentă funcțională  $X \rightarrow Y$ , cel puțin una dintre următoarele afirmații este adevărată:

- $X \rightarrow Y$  este o dependentă funcțională banală ( $Y \subseteq X$ )
- $X$  este o super cheie

Spre deosebire de primele trei forme normale, forma normală Boyce-Codd poate să nu fie obținută întotdeauna.

### Dependență funcțională

În cadrul unei relații  $r(R)$ , spunem că un set de attribute  $X$  determină funcțional un set de attribute  $Y$ , și notăm  $X \rightarrow Y$ , dacă și numai dacă fiecare valoare a lui  $X$  este unic asociată cu o valoare a lui  $Y$ . În acest caz, spunem despre  $r(R)$  că satisfacă dependentă funcțională  $X \rightarrow Y$ .

Cu alte cuvinte, dacă o valoare  $x$  a lui  $X$  este cunoscută, atunci valorile pentru attributele  $Y$  corespunzătoare valorii  $x$  pot fi determinate din ORICE tuplă din  $r(R)$  care conține valoarea  $x$  pentru attributele  $X$ .

Setul de attribute  $X$  se mai numește setul determinant, iar setul de attribute  $Y$  se mai numește setul determinat sau setul dependent.

O dependentă funcțională  $X \rightarrow Y$  se numește dependentă funcțională banală dacă  $Y$  este un subset a lui  $X$  ( $Y \subseteq X$ ).

Exemplu:

**Tabela împrumuturi**

CititorID	ISBN_carte	ID_intern_carte	Dată_împrumut	Dată_restituire
1	ISBN_A	ISBN_A#1	03/05/2016	10/05/2016
1	ISBN_B	ISBN_B#1	04/05/2016	18/05/2016
2	ISBN_A	ISBN_A#2	02/05/2016	NULL
3	ISBN_D	ISBN_D#1	17/05/2016	25/05/2016
3	ISBN_A	ISBN_A#1	20/05/2016	NULL

Tabela împrumuturi reprezintă toate cărțile împrumutate. Presupunem că un cititor poate împrumuta o carte doar o singura dată. O carte este identificată unic prin ISBN. O carte cu ISBN\_X poate fi disponibilă în mai multe exemplare, cu ID-uri interne ISBN\_X\_#1, ISBN\_X\_#1, ..., ISBN\_X\_#N.

Dependențele funcționale din tabela împrumuturi sunt:

- CititorID, ISBN\_carte, ID\_intern\_carte -> Dată\_împrumut, Dată\_restituire
- CititorID, ISBN\_carte -> ID\_intern\_carte, Dată\_împrumut, Dată\_restituire
- CititorID, ID\_intern\_carte -> ISBN\_carte, Dată\_împrumut, Dată\_restituire
- ID\_intern\_carte -> ISBN\_Carte

Din aceste dependențe funcționale, se poate observa că singurele chei candidat sunt <CititorID, ISBN\_carte> și <CititorID, ID\_intern\_carte>.

Tabela împrumuturi nu este în forma normală Boyce-Codd deoarece atributul determinant ID\_intern\_carte nu reprezintă o super cheie.

Pentru a aduce tabela în forma normală Boyce-Codd, aceasta trebuie spartă în două tabele după cum urmează:

**Tabela împrumuturi**

CititorID	ID_intern_carte	Dată_împrumut	Dată_restituire
1	ISBN_A#1	03/05/2016	10/05/2016
1	ISBN_B#1	04/05/2016	18/05/2016
2	ISBN_A#2	02/05/2016	NULL
3	ISBN_D#1	17/05/2016	25/05/2016
3	ISBN_A#1	20/05/2016	NULL

**Tabela coduri\_interne**

ID_intern_carte	ISBN_carte
ISBN_A#1	ISBN_A
ISBN_A#2	ISBN_A
ISBN_B#1	ISBN_B
ISBN_D#1	ISBN_D

Cheia primară pentru prima tabelă este  $\langle \text{CititorID}, \text{ID\_intern\_carte} \rangle$ , iar pentru a doua tablă este  $\langle \text{ID\_intern\_carte} \rangle$ . Toate dependențele funcționale respectă condițiile impuse de forma normală Boyce-Codd.

#### Pași pentru obținerea formei normale Boyce-Codd

Următorul algoritm este un algoritm simplu care funcționează în majoritatea cazurilor (*NU în toate cazurile*) și care poate fi folosit pentru a trece în forma normală Boyce-Codd:

1. Identificarea unei dependențe funcționale non-banale  $X \rightarrow Y$  care nu îndeplinește criteriul formei normale Boyce-Codd (adică  $X$  nu este o super cheie)
2. Spargerea tabelei în două tabele:
  - una care conține atributele  $XY$  (toate atributele din dependența funcțională)
  - una care conține atributele  $X$  împreună cu restul atributelor din tabela inițială (fără  $Y$ )
3. Procesul se repetă până când toate tabelele ajung în forma normală Boyce-Codd.

#### A patra formă normală

O relație este în a patra formă normală dacă și numai dacă pentru fiecare dependență multi-valoare  $X \rightarrow\!> Y$ , cel puțin una dintre următoarele afirmații este adevărată:

- $X \rightarrow\!> Y$  este o dependență multi-valoare banală ( $Y \subseteq X$  sau  $X \cup Y$  reprezintă toate atributele relației)
- $X$  este o super cheie

#### Dependențe multi-valoare

În cadrul unei relații  $r(R)$  cu un set de atribut  $W$ , spunem că un set de atribut  $X$  multi-determină un set de atribut  $Y$ , și notăm  $X \rightarrow\!> Y$ , dacă și numai dacă pentru fiecare pereche de tuple  $t1$  și  $t2$  din  $r(R)$  pentru care  $t1[X] = t2[X]$  (tuplele  $t1$  și  $t2$  au aceleași valori pentru atributul  $X$ ), există tuplele  $t3$  și  $t4$  care satisfac următoarele condiții:

- $t3[X] = t4[X] = t1[X] = t2[X]$
- $t3[Y] = t1[Y]$
- $t3[W-(X \cup Y)] = t2[W-(X \cup Y)]$
- $t4[Y] = t2[Y]$
- $t4[W-(X \cup Y)] = t1[W-(X \cup Y)]$

#### *Altfel spus:*

Presupunem că avem două tuple de forma  $(X, Y, W-(X \cup Y))$  care au următoarele valori pentru cele trei seturi de atribut:

- $t1 = (a, b, c)$
- $t2 = (a, d, e)$

Existența dependenței multi-valoare  $X \rightarrow\!> Y$  presupune existența următoarelor două tuple:

- $t3 = (a, b, e)$
- $t4 = (a, d, c)$

Exemplu:

**Tabela IT\_servicii**

<u>ITCompany</u>	<u>Locație</u>	<u>Developers</u>
Amazon	Iași	.NET
Amazon	Iași	Java
Amazon	București	.NET
Amazon	București	Java
Ness	Iași	SAP
Ness	Iași	Python
Ness	Cluj-Napoca	SAP
Ness	Cluj-Napoca	Python
Ness	Timișoara	SAP
Ness	Timișoara	Python

În exemplul de mai sus, în cazul în care pentru fiecare companie IT în parte, disponibilitatea serviciilor IT oferite nu depinde de locație (adică o companie IT oferă aceleași servicii în toate locațiile în care activează), atunci tabela NU este în a patra formă normală. Acest lucru este evident deoarece există două dependențe multi-valoare, ITCompany ->>Locație și ITCompany ->>Developers, al căror determinant este atributul ITCompany. Una dintre condițiile impuse de a patra formă normală este ca seturile de attribute determinante să fie super chei. Prin urmare, deoarece atributul ITCompany NU este super cheie, tabela nu se află în a patra formă normală. Se poate observa un nivel ridicat de redundanță în tabelă: de exemplu, în cazul companiei Ness apare de 3 ori informația că oferă servicii SAP.

Pe de alta parte, dacă semantica relației specifică că serviciile IT oferite sunt DEPENDENTE de locație, atunci tabela de mai sus ESTE în a patra formă normală, deoarece nu are dependențe multi-valoare, CHIAR DACĂ, pe moment, datele din tabelă PAR a indică prezența unor dependențe multi-valoare. Acest lucru devine clar în momentul în care se șterge orice linie din tabel, sau se inserează o linie nouă cu o altă locație și/sau un nou serviciu IT pentru oricare dintre companiile prezente în tabel.

Revenind în ipoteza inițială, dacă semantica relației specifică că serviciile IT oferite sunt INDEPENDENTE de locație (adică există dependențele multi-valoare de mai sus), pentru a aduce tabela în a patra formă normală, aceasta trebuie spartă în două tabele după cum urmează (altfel, sunt posibile anomalii – se pot inseră/șterge înregistrări astfel încât tabela să nu mai respecte dependențele multi-valoare specificate de semantica relației):

**Tabela IT\_locații**

<u>ITCompany</u>	<u>Locație</u>
Amazon	Iași
Amazon	București
Ness	Iași
Ness	Cluj-Napoca
Ness	Timișoara

**Tabela IT\_servicii**

ITCompany	Developers
Amazon	.NET
Amazon	Java
Ness	SAP
Ness	Python

### A cincea formă normală

O relație este în a cincea formă normală dacă și numai dacă fiecare dependență de tip join non-banală are la bază super chei. O dependență de tip join \*(A, B, ..., Z) are la bază super chei dacă A, B, ..., Z sunt super chei.

A cincea formă normală se mai numește și forma normală join-proiecții.

O tabelă este în a cincea formă normală dacă se află în a patra formă normală și nu mai poate fi descompusă în sub-tabele cu chei diferite.

### Dependențe de tip join

Fie relația  $r(R)$  și  $A, B, \dots, Z$  subseturi de atribute din  $R$ . Spunem că relația  $r(R)$  satisface dependența de tip join  $*(A, B, \dots, Z)$  dacă și numai dacă este egală cu join-ul proiecțiilor sale peste  $A, B, \dots, Z$ . Altfel spus:

$$r = \pi_A(r) \bowtie \pi_B(r) \bowtie \dots \bowtie \pi_Z(r)$$

O dependență de tip join este banală dacă unul dintre subseturile de atribute după care se fac proiecțiile este egal cu  $R$ .

Exemplu:

**Tabela IT\_servicii**

ITCompany	Locație	Developers
Amazon	Iași	.NET
Amazon	Iași	Java
Amazon	București	.NET
Ness	Iași	SAP
Ness	Cluj-Napoca	Python
Ness	Timișoara	SAP

În exemplul de mai sus, tabela se află în a patra formă normală (nu există dependențe multi-valoare). Totuși există o redundanță a datelor dată de următoarea regulă: *orice companie care oferă serviciul S și este prezentă într-o locație L unde acel serviciu este disponibil, trebuie să ofere serviciul S în locația L*.

Pentru a aduce tabela în a cincea formă normală, aceasta trebuie spartă în trei tabele (toate trei fiind necesare pentru reconstrucția informațiilor din tabela inițială) după cum urmează:

**Tabela IT\_locații**

<u>ITCompany</u>	<u>Locație</u>
Amazon	Iași
Amazon	București
Ness	Iași
Ness	Cluj-Napoca
Ness	Timișoara

**Tabela IT\_resurseLocație**

<u>Locație</u>	<u>Developers</u>
Iași	.NET
Iași	Java
Iași	SAP
București	.NET
Cluj-Napoca	Python
Timișoara	SAP

**Tabela IT\_servicii**

<u>ITCompany</u>	<u>Developers</u>
Amazon	.NET
Amazon	Java
Ness	SAP
Ness	Python

Eliminând redundanțele, dimensiunea datelor normalizate crește liniar, în timp ce în tabela nenormalizată, dimensiunea datelor crește multiplicativ. De exemplu, ținând cont de regula de mai sus, dacă inserăm o nouă companie IT care operează în x locații existente și oferă y servicii existente, toate aceste servicii fiind disponibile în toate cele x locații, atunci este necesar să inserăm un număr de x\*y înregistrări în tabela nenormalizată, și un număr de doar x+y înregistrări în tabelele normalize (x înregistrări în tabela IT\_locații și y înregistrări în tabela IT\_servicii).

## Joinuri

În cadrul bazelor de date relaționale, un join combină înregistrările din două sau mai multe tabele.

Standardul ANSI SQL specifică 5 tipuri de joinuri:

- Cross join
- Inner join
- Left outer join
- Right outer join
- Full outer join

Pentru a ilustra tipurile de joinuri vom folosi următoarele tabele:

**Tabela employees**

<u>EmployeeID</u>	<u>Name</u>	<u>DepartmentID</u>
1	Ioana	10
2	Marius	30
3	Anca	10
4	Dumitru	20
5	Eugen	NULL

**Tabela departments**

<u>DepatmentID</u>	<u>Name</u>
10	Marketing
20	Vânzări
30	Relații publice
40	Cercetare

### Cross join

Returnează produsul cartezian al înregistrărilor din tabelele de intrare. În exemplul de mai sus, va combina fiecare linie din tabela Employees cu fiecare linie din tabela Departments.

Folosind notația implicită:

```
SELECT * FROM employees, departments;
```

Folosind notația explicită:

```
SELECT * FROM employees CROSS JOIN departments;
```

employees. EmployeeID	employees. Name	employees. DepartmentID	departments. DepartmentID	departments. Name
1	Ioana	10	10	Marketing
2	Marius	30	10	Marketing
3	Anca	10	10	Marketing
4	Dumitru	20	10	Marketing
5	Eugen	NULL	10	Marketing
1	Ioana	10	20	Vânzări
2	Marius	30	20	Vânzări
3	Anca	10	20	Vânzări
4	Dumitru	20	20	Vânzări
5	Eugen	NULL	20	Vânzări
1	Ioana	10	30	Relații publice
2	Marius	30	30	Relații publice
3	Anca	10	30	Relații publice
4	Dumitru	20	30	Relații publice
5	Eugen	NULL	30	Relații publice
1	Ioana	10	40	Cercetare
2	Marius	30	40	Cercetare
3	Anca	10	40	Cercetare
4	Dumitru	20	40	Cercetare
5	Eugen	NULL	40	Cercetare

### Inner join

Combină fiecare linie din tabela Employees cu fiecare linie din tabela Departments în baza unor atribute care se potrivesc din perspectiva unui predicat de join (join predicate).

Folosind notația implicită:

```
SELECT * FROM employees, departments
WHERE employees.DepartmentID = departments.DepartmentID;
```

Folosind notația explicită:

```
SELECT * FROM employees INNER JOIN departments
ON employees.DepartmentID = departments.DepartmentID;
```

employees. EmployeeID	employees. Name	employees. DepartmentID	departments. DepartmentID	departments. Name
1	Ioana	10	10	Marketing
2	Marius	30	30	Relații publice
3	Anca	10	10	Marketing
4	Dumitru	20	20	Vânzări

## Equi join si Non-Equi join

Inner join se numește:

- equi join când predicatul de join folosește operatorul =
- non-equi join când predicatul de join folosește alți operatori (<, > etc.)

## Outer join

Un outer join nu necesită ca fiecare înregistrare din cele două tabele să se potrivească din perspectiva predicatului de join. Rezultatul returnat va conține toate înregistrările, chiar dacă nu s-a găsit nici o înregistrare care să se potrivească în celală tabelă.

Fie A și B cele două tabele folosite la intrarea în outer join.

Outer joinurile se împart în:

- Left outer join – întotdeauna va returna toate înregistrările din tabela A (left table), chiar dacă pentru o parte dintre acestea nu s-a găsit o potrivire cu înregistrările din tabela B, caz în care, partea cu coloanele aferente tăbelei B va conține NULL-uri.
- Right outer join – întotdeauna va returna toate înregistrările din tabela B (right table), chiar dacă pentru o parte dintre acestea nu s-a găsit o potrivire cu înregistrările din tabela A, caz în care, partea cu coloanele aferente tăbelei A va conține NULL-uri.
- Full outer join – combină efectele left outer join și right outer join. Întotdeauna va returna toate înregistrările din tabela A (left table) și din tabela B (right table), chiar dacă pentru o parte dintre acestea nu s-a găsit o potrivire cu înregistrările din tabela B, respectiv tăbelea A, caz în care, partea cu coloanele aferente tăbelei B, respectiv tăbelei A, va conține NULL-uri.

### Left outer join

Folosind sintaxa Oracle:

```
SELECT * FROM employees, departments  
WHERE employees.DepartmentID = departments.DepartmentID (+)
```

Folosind notația explicită:

```
SELECT * FROM employees LEFT OUTER JOIN departments  
ON employees.DepartmentID = departments.DepartmentID;
```

employees. EmployeeID	employees. Name	employees. DepartmentID	departments. DepartmentID	departments. Name
1	Ioana	10	10	Marketing
2	Marius	30	30	Relații publice
3	Anca	10	10	Marketing
4	Dumitru	20	20	Vânzări
5	Eugen	NULL	NULL	NULL

## Right outer join

Folosind sintaxa Oracle:

```
SELECT * FROM employees, departments  
WHERE employees.DepartmentID(+) = departments.DepartmentID
```

Folosind notația explicită:

```
SELECT * FROM employees RIGHT OUTER JOIN departments  
ON employees.DepartmentID = departments.DepartmentID;
```

employees. EmployeeID	employees. Name	employees. DepartmentID	departments. DepartmentID	departments. Name
1	Ioana	10	10	Marketing
2	Marius	30	30	Relații publice
3	Anca	10	10	Marketing
4	Dumitru	20	20	Vânzări
NULL	NULL	NULL	40	Cercetare

## Full outer join

Folosind notația explicită:

```
SELECT * FROM employees FULL OUTER JOIN departments  
ON employees.DepartmentID = departments.DepartmentID;
```

employees. EmployeeID	employees. Name	employees. DepartmentID	departments. DepartmentID	departments. Name
1	Ioana	10	10	Marketing
2	Marius	30	30	Relații publice
3	Anca	10	10	Marketing
4	Dumitru	20	20	Vânzări
5	Eugen	NULL	NULL	NULL
NULL	NULL	NULL	40	Cercetare

## Self join

Self join este un caz special de join, în care o tabelă este combinată cu ea însăși.

Fie următoarea tabelă:

**Tabela employees**

EmployeeID	Name	Age	ManagerID
1	Ioana	31	2
2	Marius	24	5
3	Anca	31	4
4	Dumitru	24	5
5	Eugen	35	NULL

De exemplu, putem folosi self join pentru a afișa toate perechile angajat – manager.

Folosind notația implicită:

```
SELECT e.Name, m.Name FROM employees e, employees m  
WHERE e.managerID = m.employeeID;
```

Folosind notația explicită:

```
SELECT e.Name, m.Name FROM employees e  
INNER JOIN employees m ON e.managerID = m.employeeID;
```

e.Name	m.Name
Ioana	Marius
Marius	Eugen
Anca	Dumitru
Dumitru	Eugen

Ca un alt exemplu, putem folosi self join pentru a afișa toate perechile de angajați care au aceeași vârstă.

Folosind notația implicită:

```
SELECT e1.Name, e2.Name, e1.Age FROM employees e1, employees e2  
WHERE e1.Age = e2.Age AND e1.employeeID != e2.employeeID;
```

Folosind notația explicită:

```
SELECT e1.Name, e2.Name, e1.Age FROM employees e1  
INNER JOIN employees e2 ON e1.Age = e2.Age  
AND e1.employeeID != e2.employeeID;
```

e1.Name	e2.Name	e1.age
Ioana	Anca	31
Marius	Dumitru	24
Anca	Ioana	31
Dumitru	Marius	24

Condiția `e1.employeeID != e2.employeeID` asigură faptul că un angajat nu este potrivit cu el însuși. Chiar și așa, se poate observa că fiecare pereche apare de două ori, în ambele sensuri. Pentru a afișa o pereche o singură dată, putem modifica condiția astfel: `e1.employeeID < e2.employeeID`. Rezultatul este următorul:

e1.Name	e2.Name	e1.age
Ioana	Anca	31
Marius	Dumitru	24

## Algoritmi folosiți pentru generarea joinurilor

Operațiile de tip join au la bază o serie de algoritmi. Printre aceștia se numără:

- Nested loop join
- Block nested loop join
- Sort-Merge join
- Hash join
- Grace hash join
- Hash anti-join
- Hash semi-join

Fiecare pagină conține o secvență (bloc) de înregistrări din cadrul unei tabele. O tabelă poate fi văzută ca o înlanțuire de astfel de pagini.

O operație I/O presupune accesarea unei întregi pagini.

### Algoritmul Nested loop join

Fiind date două tabele A și B (se poate extinde la orice număr de tabele), algoritmul este:

```
for each tuple a in A do
    for each tuple b in B do
        if a and b satisfy the join condition then
            generate a result tuple containing fields from a and b
```

Complexitate:  $O(N_A * N_B)$ , unde  $N_A$  și  $N_B$  reprezintă numărul de înregistrări din tabela A, respectiv B.

Complexitate I/O:  $O(NP_A + N_A * NP_B)$  unde  $NP_A$  și  $NP_B$  reprezintă numărul de pagini din tabela A, respectiv B.

### Algoritmul Block nested loop join

Reprezintă o generalizare a algoritmului Nested loop join.

Fiind date două tabele A și B (se poate extinde la orice număr de tabele), algoritmul este:

```

for each page Pa in A do
    for each page Pb in B do
        for each tuple a in Pa do
            for each tuple b in Pb do
                if a and b satisfy the join condition then
                    generate a result tuple
                    containing fields from a and b

```

Complexitate:  $O(N_A * N_B)$ , unde  $N_A$  și  $N_B$  reprezintă numărul de înregistrări din tabela A, respectiv B.

Complexitate I/O:  $O(NP_A + NP_A * NP_B) = O(NP_A * NP_B)$ , unde  $NP_A$  și  $NP_B$  reprezintă numărul de pagini din tabela A, respectiv B.

Optimizare: pentru a minimiza numărul de operații I/O, la fiecare pas se pot citi din tabela A un număr de pagini suficient de mare pentru a utiliza toată memoria sistemului. În acest caz, complexitatea I/O este  $O(NP_A + \text{roundUp}(NP_A/M) * NP_B)$ , unde M este memoria sistemului în pagini. Dacă numărul din pagini din tabela A este mai mic decât numărul de pagini care poate fi stocat în memoria sistemului, atunci complexitatea I/O devine:  $O(NP_A + NP_B)$ .

### Algoritmul Sort-Merge join

Fiind date două tabele A și B (se poate extinde la orice număr de tabele), și atributul de join x, algoritmul generează toate perechile de tuple pentru care x are aceeași valoare (equi-join).

```

function sortMerge(list A, list B, attribute x)
    list output = EMPTY_LIST

    list A_sorted = sort(A, x) // sortează A după atributul x
    list B_sorted = sort(B, x) // sortează B după atributul x
    list A_sublist = advance(A_sorted, x)
    list B_sublist = advance(B_sorted, x)

    while not empty(A_sublist) and not empty(B_sublist)
        if A_sublist[0].x = B_sublist[0].x // join match
            add cross product of A_sublist and B_sublist to output
            A_sublist = advance(A_sorted, x)
            B_sublist = advance(B_sorted, x)

```

```

        else if A_sublist[0].x < B_sublist[0].x
            A_sublist = advance(A_sorted, x)
        else // A_sublist[0].x > B_sublist[0].x
            B_sublist = advance(B_sorted, x)

    return output

function advance(list L_sorted, attribute x)
    list output = EMPTY_LIST
    var x_value = L_sorted[0].x
    while not empty(L_sorted) and L_sorted[0].x = x_value
        add L_sorted[0] to output
        remove L_sorted[0]
    return output

```

Complexitate:  $O(N_A \log(N_A) + N_B \log(N_B) + N_A + N_B + k) = O(N_A \log(N_A) + N_B \log(N_B) + k)$ , unde  $N_A$  și  $N_B$  reprezintă numărul de înregistrări din tabela A, respectiv B, iar  $k$  reprezintă numărul de operații pentru generarea produselor carteziene.

Complexitate I/O:  $O(NP_A + NP_B)$ , unde  $NP_A$  și  $NP_B$  reprezintă numărul de pagini din tabela A, respectiv B.

### Algoritmul Hash join

Fiind date două tabele A și B, și atributul de join x, algoritmul generează toate perechile de tuple pentru care x are aceeași valoare (equi-join).

Algoritmul conține două faze:

- Build phase – faza de construire a hash table-ului

Constă în construirea în memorie a unui hash table peste atributul x pentru tabela cu numărul cel mai mic de înregistrări. Fiecare intrare din hash table conține valoarea atributului x și identificatorul (numărul, offset-ul etc.) înregistrării în care se găsește acesta.

- Probe phase – faza de identificare a potrivirilor

Scanarea tabelei cu numărul cel mai mare de înregistrări și găsirea potrivirilor prin căutarea atributului x al fiecărei înregistrări în hash table-ul construit în pasul anterior.

Pseudocodul algoritmului:

Presupunem că A este tabela cu numărul cel mai mic de înregistrări, iar B tabela cu numărul cel mai mare de înregistrări.

```

for each tuple a in A
    add a to the in-memory hash table

if hash table not empty
    for each tuple b in B
        for each tuple match a of b.x in the hash table
            add the join of a and b to the output

    clear hash table

```

Complexitate:  $O(N_A + N_B + k)$ , unde  $N_A$  și  $N_B$  reprezintă numărul de înregistrări din tabela A, respectiv B, iar  $k$  reprezintă numărul de operații pentru generarea joinurilor dintre tuple.

Complexitate I/O:  $O(NP_A + NP_B)$ , unde  $NP_A$  și  $NP_B$  reprezintă numărul de pagini din tabela A, respectiv B.

Dacă hash table-ul generat este prea mare pentru a putea fi ținut în memorie, atunci algoritmul poate fi modificat astfel:

```

for each tuple a in A
    add a to the in-memory hash table

    if memory is full
        for each tuple b in B
            for each tuple match a of b.x in the hash table
                add the join of a and b to the output

        clear hash table

if hash table not empty
    for each tuple b in B
        for each tuple match a of b.x in the hash table
            add the join of a and b to the output

    clear hash table

```

Complexitate:  $O(N_A + \text{roundUp}(\text{sizeof(FULL hash table})/M) * N_B + k)$ , unde  $N_A$  și  $N_B$  reprezintă numărul de înregistrări din tabela A, respectiv B, iar  $k$  reprezintă numărul de operații pentru generarea joinurilor dintre tuple. Dacă hash table-ul poate fi stocat în întregime în memoria sistemului, atunci complexitatea devine:  $O(N_A + N_B + k)$ .

Complexitate I/O:  $O(NP_A + \text{roundUp}(\text{sizeof(FULL hash table})/M) * NP_B)$ , unde  $M$  este memoria sistemului. Dacă hash table-ul poate fi stocat în întregime în memoria sistemului, atunci complexitatea I/O devine:  $O(NP_A + NP_B)$ .

## Algoritmul Grace hash join

Reprezintă o îmbunătățire a algoritmului Hash join. În cazul algoritmului Hash join, dacă hash table-ul construit din tabela A nu începe în totalitate în memorie, tabela B este rescanată de mai multe ori. Acest lucru este evitat în algoritmul Grace hash join.

Algoritmul partionează tabelele A și B folosind o funcție hash peste atributul x. Partițiile astfel generate sunt stocate pe hard disk. Pentru fiecare pereche de partiții care au același hash, se aplică algoritmul simplu Hash join. Este evident că toate potrivirile dintre A și B, dacă există, vor fi generate doar din perechile de partiții care au același hash peste atributul x.

Rezultatul final reprezintă o concatenare a rezultatelor parțiale obținute din toate perechile de partiții.

Dacă pentru o pereche de partiții, hash table-ul generat nu începe în memorie, algoritmul este aplicat recursiv (pentru această pereche) cu o altă funcție de hash peste atributul x.

Complexitate:  $O(N_A + N_B + k)$ , unde  $N_A$  și  $N_B$  reprezintă numărul de înregistrări din tabela A, respectiv B, iar  $k$  reprezintă numărul de operații pentru generarea joinurilor dintre tuple.

Complexitate I/O:  $O(NP_A + NP_B)$ , unde  $NP_A$  și  $NP_B$  reprezintă numărul de pagini din tabela A, respectiv B.

## Algoritmul Hash anti-join

Fiind date două tabele A și B, și atributul de anti-join x, algoritmul selectează tuplele din A pentru care valoarea atributului x NU se găsește în tabela B.

... FROM A NOT IN B

Ex: `select * from A where x NOT IN (select distinct x from B)`

### Hash left anti-join

Algoritmul conține două faze:

- Construirea hash table-ului peste atributul x pentru tabela B. Fiecare intrare din hash table conține valoarea atributului x.
- Scanarea tabelei A și selectarea tuplelor al căror atribut x nu se găsește în hash table-ul construit în pasul anterior.

Este mult mai eficient dacă tabela B conține un număr mai mic de înregistrări decât tabela A.

Complexitate: **temă de gândire**

Complexitate I/O: **temă de gândire**

### Hash right anti-join

Algoritmul conține două faze:

- Construirea hash table-ului peste atributul x pentru tabela A. Fiecare intrare din hash table conține valoarea atributului x și identificatorul (numărul, offset-ul etc.) înregistrării în care se găsește acesta.
- Scanarea tabelei B și ștergerea tuturor intrărilor din hash table-ul construit în pasul anterior care se potrivesc cu atributul x.

Rezultatul este dat de intrările rămase în hash table.

Este mult mai eficient dacă tabela A conține un număr mai mic de înregistrări decât tabela B.

Complexitate: **temă de gândire**

Complexitate I/O: **temă de gândire**

### Algoritmul Hash semi-join

Fiind date două tabele A și B, și atributul de semi-join x, algoritmul selectează tuplele din A pentru care valoarea atributului x se găsește în tabela B.

... FROM A IN B

Ex: select \* from A where x IN (select distinct x from B)

#### Hash left semi-join

Algoritmul conține două faze:

- Construirea hash table-ului peste atributul x pentru tabela B. Fiecare intrare din hash table conține valoarea atributului x.
- Scanarea tabelei A și selectarea tuplelor al căror atribut x se găsește în hash table-ul construit în pasul anterior.

Este mult mai eficient dacă tabela B conține un număr mai mic de înregistrări decât tabela A.

Complexitate: **temă de gândire**

Complexitate I/O: **temă de gândire**

#### Hash right semi-join

Algoritmul conține două faze:

- Construirea hash table-ului peste atributul x pentru tabela A. Fiecare intrare din hash table conține valoarea atributului x și identificatorul (numărul, offset-ul etc.) înregistrării în care se găsește acesta.
- Scanarea tabelei B și selectarea tuturor intrărilor din hash table-ul construit în pasul anterior care se potrivesc cu atributul x, **urmată de ștergerea acestora din hash table**.

Fiecare înregistrare din A, pentru care există potrivire, este selectată doar o singură dată deoarece, la prima potrivire, este scoasă din hash table.

Este mult mai eficient dacă tabela A conține un număr mai mic de înregistrări decât tabela B.

Complexitate: **temă de gândire**

Complexitate I/O: **temă de gândire**

## ACID (Atomicity, Consistency, Isolation, Durability)

ACID reprezintă un set de proprietăți tranzacționale care garantează o procesare coerentă a tranzacțiilor.

În cadrul bazelor de date, o tranzacție reprezintă un set de operații de citire/scriere/actualizare a datelor. Acest set de operații poate fi văzut ca o singură operație logică asupra datelor.

De exemplu, transferul de fonduri între două conturi, deși implică accesarea și modificarea a cel puțin două înregistrări din baza de date (debitarea unui cont și creditarea altuia), reprezintă o singură operație logică asupra datelor, fiind executat într-o singură tranzacție.

### Atomicitate (atomicity)

Toate operațiile executate asupra datelor în cadrul unei tranzacții sunt aplicate atomic. Cu alte cuvinte, ori toate operațiile sunt executate, ori nici una dintre acestea nu este executată.

O tranzacție este indivizibilă, având la bază conceptul "totul sau nimic": dacă o parte dintr-o operație unei tranzacții eșuează, atunci toată tranzacția eșuează, iar baza de date nu este modificată în nici un fel.

Un sistem de baze de date care suportă atomicitate, trebuie să garanteze această proprietate în orice situație (întreruperi de curent, erori de aplicație etc.).

Implementare:

- Menținere unei copii a datelor înainte de modificare
- Jurnalizare (journaling) – log tranzacțional (transaction log)

### Coerență (consistency)

Presupune faptul că orice tranzacție schimbă starea sistemului dintr-o stare validă în altă stare validă. Toate datele dintr-o bază de date care suportă coerență, trebuie să fie valide din punct de vedere al tuturor regulilor definite asupra tabelelor și datelor din tabele. Deși acest lucru garantează validitatea datelor relativ la regulile definite, nu protejează împotriva erorilor de programare care pot genera erori semantice în cadrul datelor.

### Coerență eventuală (Eventual consistency)

Este un model de coerență a datelor utilizat în cadrul sistemelor distribuite cu scopul de a obține o mare disponibilitate (high availability) a datelor.

Acest model garantează că, dacă nu se mai fac actualizări asupra datelor, în cele din urmă, toate nodurile din sistemul distribuit vor returna aceleași date (actualizările sunt propagate în tot sistemul). Un sistem distribuit care a ajuns în această stare se spune că a ajuns la convergență sau că a ajuns la convergență replicilor.

## Izolare (isolation)

Această proprietate asigură faptul că starea sistemului rezultată după execuția în paralel a tranzacțiilor este aceeași cu starea sistemului care ar fi rezultat dacă aceste tranzacții ar fi fost executate în mod serial (una după alta).

Există mai multe niveluri de izolare. Cu cât nivelul de izolare este mai relaxat, efectele unor tranzacții incomplete pot fi vizibile în cadrul altor tranzacții.

Un nivel relaxat de izolare presupune un nivel ridicat de accesare simultană a datelor din tranzacții diferite. Totuși, acest lucru presupune creșterea vizibilității unor efecte nedorite, cum ar fi citirea unor date incomplete sau suprascrierea datelor între tranzacții.

Un nivel ridicat de izolare presupune limitarea vizibilității efectelor nedorite cu costul blocării temporare a tranzacțiilor în cazul în care acestea lucrează pe aceleși date. Cu alte cuvinte, numărul de tranzacții care pot fi executate simultan în sistem, și care accesează aceleși date, scade.

Prin urmare, relaxarea nivelului de izolare oferă performanțe crescute, dar presupune o abatere de la principiul de serializabilitate al tranzacțiilor, acestea putând accesa date incomplete care aparțin altor tranzacții care se execută simultan.

## Two phase locking

Two phase locking este o metodă des utilizată de control al concurenței tranzacțiilor în care, **înainte** de a accesa datele, o tranzacție trebuie mai întâi să obțină lock-uri pentru acele date. Această metodă garantează serializabilitatea tranzacțiilor.

Cele două faze sunt:

- faza de expansiune – noi lock-uri sunt obținute fără a elibera nici un alt lock deja obținut. Prin urmare, numărul de lock-uri obținute în timpul acestei faze este strict crescător.
- faza de contracție – lockurile obținute sunt eliberate și nici un alt lock nu mai este obținut. Prin urmare, numărul de lock-uri scade strict până la 0 în timpul acestei faze.

Există două tipuri majore de lock-uri:

- read locks (lock-uri de citire) – sunt lock-uri de tip shared
- write locks (lock-uri de scriere) – sunt lock-uri de tip exclusive

Interacțiunile dintre cele două tipuri de lock-uri:

- Un lock de write pe un obiect **blochează** obținerea din altă tranzacție a unui lock de write pe același obiect. Cel de-al doilea lock va putea fi luat după ce primul lock este eliberat.
- Un lock de write pe un obiect **blochează** obținerea din altă tranzacție a unui lock de read pe același obiect. Cel de-al doilea lock va putea fi luat după ce primul lock este eliberat.
- Un lock de read pe un obiect **blochează** obținerea din altă tranzacție a unui lock de write pe același obiect. Cel de-al doilea lock va putea fi luat după ce primul lock este eliberat.
- Un lock de read pe un obiect **îngăduie** obținerea din altă tranzacție a unui lock de read pe același obiect. Cel de-al doilea lock este luat imediat ce este cerut.

Dacă un lock blochează un alt lock atunci cele două lock-uri sunt incompatibile (marcat cu **X** în tabela de mai jos), altfel, acestea sunt compatibile.

Tip lock	read lock	write lock
read lock		<b>X</b>
write lock	<b>X</b>	<b>X</b>

Blocarea obținerii unui lock presupune oprirea temporară a tranzacției care solicită lock-ul, până când acesta este eliberat de tranzacția care a obținut lock-ul. După eliberare, tranzacția oprită este repornită, lock-ul este preluat de aceasta, iar execuția ei continuă din punctul în care a fost oprită.

În cazul acestei metode pot apărea dead-lockuri: două tranzacții se blochează reciproc, așteptându-se una pe cealaltă.

### Snapshot isolation

În cadrul procesării tranzacțiilor, snapshot isolation reprezintă o garanție a faptului că toate citirile din cadrul unei tranzacții vor vedea un snapshot coherent al bazei de date. Implementările snapshot isolation furnizează ultimele valori committed de dinaintea pornirii tranzacției. Tranzacția va fi executată cu succes dacă orice modificări aduse asupra datelor în cadrul acesteia nu intră în conflict cu modificările aduse de alte tranzacții care rulează concurrent.

### Anomalia write skew

Presupunem ca avem un client al unei bănci cu două conturi care pot fi descoperite, dar pentru care există o constrângere de tipul  $\text{Account1} + \text{Account2} \geq 0\$$ .

Time	Object 1	Object 2
0	Account1 = 100\$	Account2 = 100\$
1	T1 (retragere 200\$ din Account1): Account1=-100\$	T2 (retragere 100\$ din Account2): Account2=0\$

La momentul 0:  $\text{Account1} + \text{Account2} = 200\$$ .

La momentul 1, cele două tranzacții T1 și T2 își încep concurrent execuția. Fiecare dintre acestea vede un snapshot al bazei de date de la momentul 0. T1 retrage 200\$ din Account1, iar T2 retrage 100\$ din Account2. Deoarece fiecare tranzacție lucrează pe snapshotul de la momentul 0, constrângerea  $\text{Account1} + \text{Account2} \geq 0\$$  este validă la sfârșitul fiecărei tranzacții (pentru T1,  $\text{Account1} + \text{Account2} = 0\$$ ; pentru T2:  $\text{Account1} + \text{Account2} = 100\$$ ). Mai mult, actualizările făcute de T1 și T2 sunt pe seturi de date disjuncte (Account1, respectiv Account2), prin urmare nu apar conflicte la commit. Totuși, după commitul cu succes al celor două tranzacții, suma  $\text{Account1} + \text{Account2} = -100\$$ , ceea ce reprezintă o încălcare a constrângerii initiale.

Soluții:

- Crearea unui alt obiect, Object 3, care să conțină suma celor două conturi și care să fie actualizat atât de T1 cât și de T2. Acest lucru va duce la conflict în momentul commitului pentru una dintre tranzacții, deoarece Object 3 va fi detectat ca modificat între timp.

- Modificarea "falsă" a Account2 din T1 cu Account2 = Account2, și similar, a Account1 din T2 cu Account1 = Account1. Acest lucru va duce la conflict în momentul commitului pentru una dintre tranzacții, deoarece contul care se dorește a fi actualizat va fi detectat ca modificat între timp.

### Multiversion concurrency control

Multiversion concurrency control implementează snapshot isolation. Este o altă metodă de control al concurenței tranzacțiilor care nu folosește lock-uri. Această metodă folosește snapshot-uri, iar fiecare tranzacție vede un snapshot al bazei de date de la un anumit moment. Schimbările asupra datelor efectuate în tranzacția curentă devin vizibile altor tranzacții doar după ce aceasta este committed.

În momentul în care se actualizează un obiect, noua valoare **nu** suprascrie vechea valoarea, ci o nouă versiune a acestui obiect este stocată în baza de date. La un moment dat pot exista mai multe versiuni ale aceluiași obiect, dar numai una dintre acestea este și cea mai actuală. Prin urmare, tranzacțiile pot vedea aceleși date, citind o anumită versiune a acestora, chiar dacă aceste date s-au modificat între timp în alte tranzacții concurente.

Folosind mecanismul de multiversion, tranzacțiile care citesc date și cele care scriu date sunt izolate unele de altele fără a fi nevoie de lock-uri. Fiecare scriere generează o nouă versiune a datelor, în timp ce citirile concurente accesează o versiune anterioară a acestora.

O tranzacție poate actualiza cu succes un obiect doar dacă acesta nu a fost modificat între timp de o altă tranzacție. În caz contrar, tranzacția curentă eșuează deoarece există deja noi versiuni ale datelor în baza de date, iar actualizările curente se bazează pe date învechite.

Time	Object 1	Object 2	Object 3
0	T0: {Ana, 23}	T0: {Marius, 18}	
1	T1: {Ana-Maria, 24}		
2		T2: delete	T2: {Ionuț, 22}

O tranzacție Tx care pornește imediat după ce T1 a fost committed dar înainte ca T2 să fie committed, și durează mai mult timp, va citi la momentul 3: Object 1 = {Ana-Maria, 24}, Object2 = {Marius, 18}, iar Object 3 nu va fi vizibil. Aceasta deoarece Tx lucrează pe un snapshot al bazei de date aşa cum era în momentul în care această tranzacție a fost pornită.

### Optimistic concurrency control

Optimistic concurrency control este o metodă de control al concurenței tranzacțiilor care nu folosește nici lock-uri și nici snapshot-uri. Această metodă presupune că un număr relativ mare de tranzacții care se execută simultan nu interferează unele cu altele.

La fiecare commit, se verifică dacă datele care se doresc a fi actualizate de tranzacția curentă au fost modificate de tranzacții concurente, caz în care tranzacția curentă eșuează. Dacă procentul de tranzacții eșuate este mic, această metodă oferă performanțe superioare față de alte metode de control al concurenței tranzacțiilor.

## Niveluri de izolare

Majoritatea SGBD-urilor (Sistemelor de Gestire a Bazelor de Date) oferă posibilitatea specificării nivelului de izolare a tranzacțiilor.

Nivelurile de izolare sunt:

- SERIALIZABLE
- REPEATABLE READ
- READ COMMITTED
- READ UNCOMMITTED

### SERIALIZABLE

Reprezintă cel mai restrictiv nivel de izolare a tranzacțiilor.

În cazul folosirii lock-urilor, sunt achiziționate lock-uri de read, write și range locks.

- lock-urile de tip write vor fi eliberate la sfârșitul tranzacției
- lock-urile de tip read vor fi eliberate la sfârșitul tranzacției
- lock-urile de tip range lock vor fi eliberate la sfârșitul tranzacției

### REPEATABLE READ

În cazul folosirii lock-urilor, sunt achiziționate lock-uri de read și write.

- lock-urile de tip write vor fi eliberate la sfârșitul tranzacției
- lock-urile de tip read vor fi eliberate la sfârșitul tranzacției
- lock-urile de tip range lock nu sunt folosite

Anomalii care pot apărea: phantom reads.

### READ COMMITTED

În cazul folosirii lock-urilor, sunt achiziționate lock-uri de read și write.

- lock-urile de tip write vor fi eliberate la sfârșitul tranzacției
- lock-urile de tip read vor fi eliberate imediat ce instrucțiunea SELECT aferentă s-a terminat
- lock-urile de tip range lock nu sunt folosite

Anomalii care pot apărea: non-repeatable reads, phantom reads.

### READ UNCOMMITTED

Reprezintă cel mai permisiv nivel de izolare a tranzacțiilor.

- lock-urile de tip write vor fi eliberate la sfârșitul tranzacției
- lock-urile de tip read nu sunt folosite
- lock-urile de tip range lock nu sunt folosite

Anomalii care pot apărea: dirty reads, non-repeatable reads, phantom reads.

ANSI/ISO standard SQL 92 descrie trei anomalii de citire în contextul în care o tranzacție T1 citește datele pe care o tranzacție concurrentă T2 le poate schimba.

Presupunem că avem următoarea tabelă:

**Tabela employees**

<b>id</b>	<b>name</b>	<b>salary</b>
<b>1</b>	Ana	2600
<b>2</b>	Ionuț	3200

### 1. Dirty reads

Un dirty read apare atunci când o tranzacție poate citi date care au fost modificate de o tranzacție concurrentă și care nu au fost încă committed.

Transaction T1	Transaction T2
<pre>SELECT salary FROM employees WHERE id = 1; /* returns 2600 */  SELECT salary FROM employees WHERE id = 1; /* returns 3000 */</pre>	<pre>UPDATE employees SET salary = 3000 WHERE id = 1; /* no commit */  ROLLBACK;</pre>

Al doilea SELECT din tranzacția T1 returnează valori care nici măcar nu vor fi committed de tranzacția T2, deoarece aceasta face ROLLBACK la sfârșit.

Această anomalie apare la nivelul de izolare: READ UNCOMMITTED.

### 2. Non-repeatable reads

Un non-repeatable read apare atunci când pe parcursul unei tranzacții o înregistrare este preluată de două ori, iar cel puțin o parte dintre valorile acestor tranzacții diferă.

Transaction T1	Transaction T2
<pre>SELECT * FROM employees WHERE id = 1; /* returns {1, Ana, 2600} */  SELECT * FROM employees WHERE id = 1; /* returns {1, Ana, 3000} */</pre>	<pre>UPDATE employees SET salary = 3000 WHERE id = 1; COMMIT;</pre>

Al doilea SELECT din tranzacția T1 returnează o valoare diferită a salariului față de primul SELECT din cadrul aceleiași tranzacții, pentru același angajat.

Această anomalie apare la nivelurile de izolare: READ UNCOMMITTED, READ COMMITTED.

### 3. Phantom reads

Un phantom read apare atunci când pe parcursul unei tranzacții două SELECT-uri identice sunt executate, iar colecțiile de înregistrări returnate sunt diferite.

Transaction T1	Transaction T2
<pre>SELECT * FROM employees WHERE salary BETWEEN 1000 and 5000; /* returns {1, Ana, 2600} {2, Ionuț, 3200} */</pre> <pre>SELECT * FROM employees WHERE salary BETWEEN 1000 and 5000; /* returns {1, Ana, 2600} {2, Ionuț, 3200} {3, Vlad, 4000} */</pre>	<pre>INSERT INTO employees VALUES (3, 'Vlad', 4000); COMMIT;</pre>

Al doilea SELECT din tranzacția T1 returnează o colecție de înregistrări diferită de colecția de înregistrări returnată de primul SELECT din cadrul aceleiași tranzacții, cele două SELECT-uri fiind identice.

Această anomalie apare la nivelurile de izolare: READ UNCOMMITTED, READ COMMITTED, REPEATABLE READ.

#### Posibilitatea apariției anomaliei în funcție de nivelul de izolare

Isolation level	dirty reads	non-repeatable reads	phantom reads
READ UNCOMMITTED	DA	DA	DA
READ COMMITTED	-	DA	DA
REPEATABLE READ	-	-	DA
SERIALIZABLE	-	-	-

#### Eliberare lock-uri în funcție de nivelul de izolare

Isolation level	write locks	read locks	range lock
READ UNCOMMITTED	la sfârșit	-	-
READ COMMITTED	la sfârșit	imediat	-
REPEATABLE READ	la sfârșit	la sfârșit	-
SERIALIZABLE	la sfârșit	la sfârșit	la sfârșit

imediat = imediat după execuția statement-ului curent

la sfârșit = la sfârșitul tranzacției

## Durabilitate (durability)

Odată ce o tranzacție a fost efectuată (committed), efectele acesteia devin permanente (chiar dacă baza de date își încetează execuția sau dacă curentul este întrerupt). Pentru a asigura acest fapt, efectele tranzacțiilor trebuie stocate într-o memorie non-volatile.

### Exemplu:

Fie tabela Test cu trei coloane, id, a și b, și o constrângere de integritate de tipul  $a+b=100$ .

```
CREATE TABLE Test (
    id NUMERIC(2) PRIMARY KEY,
    a NUMERIC(2),
    b NUMERIC(2),
    CONSTRAINT c CHECK (a+b=100)
);
```

```
INSERT INTO Test VALUES (1,50,50);
```

### *Consistency failure*

```
BEGIN TRANSACTION
    UPDATE Test SET a=30 WHERE id=1;
    COMMIT
```

Tranzacția poate fi aplicată atomic, dar proprietatea de coerență (verificată la sfârșitul tranzacției) nu este îndeplinită. Prin urmare, întreaga tranzacție va fi anulată (rolled back).

### *Isolation failure*

Presupunem că avem două tranzacții, T1 și T2, care se execută în același timp. Fiecare tranzacție modifică înregistrarea cu id-ul 1.

Presupunem că T1 transferă 10 din coloana a în coloana b, iar T2 transferă 20 din coloana b în coloana a.

Dacă tranzacțiile se execută secvențial, izolarea este menținută, chiar și în cazul în care T1 eșuează înainte de a executa a doua operație (T1 va fi anulată, iar T2 va vedea datele inițiale):

```

INITIAL: (id:1, a:50, b:50)

BEGIN TRANSACTION T1

    COMMITTED VALUES: (id:1, a:50, b:50)

    T1: a=a-10 -> (id:1, a:40, b:50)

    T1: b=b+10 -> (id:1, a:40, b:60)

COMMIT T1

BEGIN TRANSACTION T2

    COMMITTED VALUES: (id:1, a:40, b:60)

    T2: b=b-20 -> (id:1, a:40, b:40)

    T2: a=a+20 -> (id:1, a:60, b:40)

COMMIT T2

```

În cazul în care sistemul execută tranzacțiile în paralel, o posibilă execuție poate fi:

```

INITIAL: (id:1, a:50, b:50)

BEGIN TRANSACTION T1

    COMMITTED VALUES: (id:1, a:50, b:50)

    T1: a=a-10 -> (id:1, a:40, b:50)

BEGIN TRANSACTION T2

    COMMITTED VALUES: (id:1, a:50, b:50)

    T2: b=b-20 -> (id:1, a:50, b:30)

    T2: a=a+20 -> (id:1, a:70, b:30)

COMMIT T2

    T1: b=b+10 -> (id:1, a:40, b:60)

COMMIT T1

```

*În cazul de mai sus, fiecare tranzacție lucrează cu datele (COMMITTED VALUES) preluate din tabelă la începutul tranzacției (**snapshot isolation**), independent de COMMIT-urile altor tranzacții care pot apărea între timp.*

În cazul în care T1 eșuează înainte de a executa a doua operație, T1 va fi anulată (rolled back). Dacă roll back-ul presupune resetarea valorii lui a la 50 (așa cum era înaintea tranzacției T1), va apărea o eroare de coerență a datelor, deoarece tranzacția T2 a modificat între timp tabela cu valorile (id:1, a:70, b:30).

*Durability failure*

```
BEGIN TRANSACTION  
  
    UPDATE Test SET a=30 WHERE id=1;  
  
    UPDATE Test SET b=70 WHERE id=1;  
  
COMMIT
```

Dacă utilizatorul primește confirmarea execuției tranzacției, acesta presupune că modificările sunt permanente. Totuși, dacă actualizările sunt încă în buffer-ul sistemului și nu au fost stocate pe hard disk, o întrerupere a curentului va duce la pierderea acestora.

Table 9-2 Conflicting Writes and Lost Updates in a READ COMMITTED Transaction

Session 1	Session 2	Explanation
<pre>SQL&gt; SELECT last_name, salary   FROM employees WHERE last_name   IN ('Banda', 'Greene', 'Hintz');  LAST_NAME      SALARY ----- Banda          6200 Greene         9500</pre>		Session 1 queries the salaries for Banda, Greene, and Hintz. No employee named Hintz is found.
<pre>SQL&gt; UPDATE employees SET salary = 7000 WHERE last_name = 'Banda';</pre>		Session 1 begins a transaction by updating the Banda salary. The default isolation level for transaction 1 is READ COMMITTED.
	<pre>SQL&gt; SET TRANSACTION ISOLATION LEVEL READ COMMITTED;</pre>	Session 2 begins transaction 2 and sets the isolation level explicitly to READ COMMITTED.
	<pre>SQL&gt; SELECT last_name, salary   FROM employees WHERE last_name IN   ('Banda', 'Greene', 'Hintz');  LAST_NAME      SALARY ----- Banda          6200 Greene         9500</pre>	Transaction 2 queries the salaries for Banda, Greene, and Hintz. Oracle Database uses read consistency to show the salary for Banda before the uncommitted update made by transaction 1.
	<pre>SQL&gt; UPDATE employees SET salary = 9900 WHERE last_name = 'Greene';</pre>	Transaction 2 updates the salary for Greene successfully because transaction 1 locked only the Banda row (see "Row Locks (TX)").
<pre>SQL&gt; INSERT INTO employees (employee_id, last_name, email, hire_date, job_id) VALUES (210, 'Hintz', 'JHINTZ', SYSDATE, 'SH_CLERK');</pre>		Transaction 1 inserts a row for employee Hintz, but does not commit.
	<pre>SQL&gt; SELECT last_name, salary   FROM employees WHERE last_name IN   ('Banda', 'Greene', 'Hintz');  LAST_NAME      SALARY ----- Banda          6200 Greene         9900</pre>	<p>Transaction 2 queries the salaries for employees Banda, Greene, and Hintz.</p> <p>Transaction 2 sees its own update to the salary for Greene. Transaction 2 does not see the uncommitted update to the salary for Banda or the insertion for Hintz made by transaction 1.</p>
	<pre>SQL&gt; UPDATE employees SET salary = 6300 WHERE last_name = 'Banda'; -- prompt does not return</pre>	Transaction 2 attempts to update the row for Banda, which is currently locked by transaction 1, creating a conflicting write. Transaction 2 waits until transaction 1 ends.
<pre>SQL&gt; COMMIT;</pre>		Transaction 1 commits its work, ending the transaction.
	<pre>1 row updated.  SQL&gt;</pre>	The lock on the Banda row is now released, so transaction 2 proceeds with its update to the salary for Banda.
	<pre>SQL&gt; SELECT last_name, salary   FROM employees WHERE last_name IN   ('Banda', 'Greene', 'Hintz');  LAST_NAME      SALARY ----- Banda          6300 Greene         9900 Hintz</pre>	Transaction 2 queries the salaries for employees Banda, Greene, and Hintz. The Hintz insert committed by transaction 1 is now visible to transaction 2. Transaction 2 sees its own update to the Banda salary.
	<pre>SQL&gt; COMMIT;</pre>	Transaction 2 commits its work, ending the transaction.
<pre>SQL&gt; SELECT last_name, salary   FROM employees WHERE last_name   IN ('Banda', 'Greene', 'Hintz');  LAST_NAME      SALARY ----- Banda          6300 Greene         9900 Hintz</pre>	Session 1 queries the rows for Banda, Greene, and Hintz. The salary for Banda is 6300, which is the update made by transaction 2. The update of Banda's salary to 7000 made by transaction 1 is now "lost."	

Table 9-3 Read Consistency and Serialized Access Problems in Serializable Transactions

Session 1	Session 2	Explanation
<pre>SQL&gt; SELECT last_name, salary   FROM employees WHERE last_name     IN ('Banda', 'Greene', 'Hintz');  LAST_NAME      SALARY -----  ----- Banda          6200 Greene         9500</pre>		Session 1 queries the salaries for Banda, Greene, and Hintz. No employee named Hintz is found.
<pre>SQL&gt; UPDATE employees SET salary = 7000 WHERE last_name = 'Banda';</pre>		Session 1 begins transaction 1 by updating the Banda salary. The default isolation level for is READ COMMITTED.
	<pre>SQL&gt; SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;</pre>	Session 2 begins transaction 2 and sets it to the SERIALIZABLE isolation level.
	<pre>SQL&gt; SELECT last_name, salary   FROM employees WHERE last_name     IN ('Banda', 'Greene', 'Hintz');  LAST_NAME      SALARY -----  ----- Banda          6200 Greene         9500</pre>	Transaction 2 queries the salaries for Banda, Greene, and Hintz. Oracle Database uses read consistency to show the salary for Banda before the uncommitted update made by transaction 1.
	<pre>SQL&gt; UPDATE employees SET salary = 9900 WHERE last_name = 'Greene';</pre>	Transaction 2 updates the Greene salary successfully because only the Banda row is locked.
<pre>SQL&gt; INSERT INTO employees (employee_id, last_name, email, hire_date, job_id) VALUES (210, 'Hintz', 'JHINTZ', SYSDATE, 'SH_CLERK');  SQL&gt; COMMIT;</pre>		Transaction 1 inserts a row for employee Hintz.
<pre>SQL&gt; SELECT last_name, salary   FROM employees WHERE last_name     IN ('Banda', 'Greene', 'Hintz');  LAST_NAME      SALARY -----  ----- Banda          7000 Greene         9500 Hintz          9900</pre>	<pre>SQL&gt; SELECT last_name, salary   FROM employees WHERE last_name IN ('Banda', 'Greene', 'Hintz');  LAST_NAME      SALARY -----  ----- Banda          6200 Greene         9900</pre>	Session 1 queries the salaries for employees Banda, Greene, and Hintz and sees changes committed by transaction 1. Session 1 does not see the uncommitted Greene update made by transaction 2.
	<pre>COMMIT;</pre>	Transaction 2 queries the salaries for employees Banda, Greene, and Hintz. Oracle Database read consistency ensures that the Hintz insert and Banda update committed by transaction 1 are <i>not</i> visible to transaction 2. Transaction 2 sees its own update to the Banda salary.
		Transaction 2 commits its work, ending the transaction.
<pre>SQL&gt; SELECT last_name, salary   FROM employees WHERE last_name     IN ('Banda', 'Greene', 'Hintz');  LAST_NAME      SALARY -----  ----- Banda          7000 Greene         9900 Hintz          9900</pre>		Both sessions query the salaries for Banda, Greene, and Hintz. Each session sees all committed changes made by transaction 1 and transaction 2.
<pre>SQL&gt; UPDATE employees SET salary = 7100 WHERE last_name = 'Hintz';</pre>		Session 1 begins transaction 3 by updating the Hintz salary. The default isolation level for transaction 3 is READ COMMITTED.
	<pre>SQL&gt; SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;</pre>	Session 2 begins transaction 4 and sets it to the SERIALIZABLE isolation level.
	<pre>SQL&gt; UPDATE employees SET salary = 7200 WHERE last_name = 'Hintz'; -- prompt does not return</pre>	Transaction 4 attempts to update the salary for Hintz, but is blocked because transaction 3 locked the Hintz row (see "Row Locks (TX)"). Transaction 4 queues behind transaction 3.
<pre>SQL&gt; COMMIT;</pre>		Transaction 3 commits its update of the Hintz salary, ending the transaction.

	<pre>UPDATE employees SET salary = 7200 WHERE last_name = 'Hintz' * ERROR at line 1: ORA-08177: can't serialize access for this transaction</pre>	The commit that ends transaction 3 causes the Hintz update in transaction 4 to fail with the ORA-08177 error. The problem error occurs because transaction 3 committed the Hintz update <i>after</i> transaction 4 began.
	<pre>SQL&gt; ROLLBACK;</pre>	Session 2 rolls back transaction 4, which ends the transaction.
	<pre>SQL&gt; SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;</pre>	Session 2 begins transaction 5 and sets it to the SERIALIZABLE isolation level.
	<pre>SQL&gt; SELECT last_name, salary   FROM employees WHERE last_name     IN ('Banda', 'Greene', 'Hintz');  LAST_NAME      SALARY -----  ----- Banda          7100 Greene         9500 Hintz          7100</pre>	Transaction 5 queries the salaries for Banda, Greene, and Hintz. The Hintz salary update committed by transaction 3 is visible.
	<pre>SQL&gt; UPDATE employees SET salary = 7200 WHERE last_name = 'Hintz'; 1 row updated.</pre>	Transaction 5 updates the Hintz salary to a different value. Because the Hintz update made by transaction 3 committed <i>before</i> the start of transaction 5, the serialized access problem is avoided.
	<pre>SQL&gt; COMMIT;</pre>	<b>Note:</b> If a different transaction updated and committed the Hintz row after transaction transaction 5 began, then the serialized access problem would occur again.
		Session 2 commits the update without any problems, ending the transaction.

## Commit în două faze (Two-phase commit protocol – 2PC)

Protocolul 2PC garantează atomicitatea unei tranzacții distribuite. Acest protocol are la bază un algoritm distribuit care coordonează mai multe noduri/procese participante pentru a garanta atomicitatea unei tranzacții distribuite care afectează mai multe baze de date simultan. 2PC presupune existența unui proces coordonator și a unui număr de procese participante.

Protocolul se desfășoară în două faze:

- faza de commit request (sau faza de votare) – coordonatorul cere participantilor să pre-finalizeze tranzacția. Fiecare participant răspunde/votează cu 'da' sau 'nu' dacă pre-commit-ul local a fost făcut cu succes.
- faza de commit – dacă **TOTI** participantii au votat 'da' atunci coordonatorul cere tuturor participantilor să aplique local commit-ul final, iar tranzacția globală este committed. În caz contrar, dacă cel puțin un participant a votat 'nu', atunci coordonatorul cere tuturor participantilor să efectueze un rollback local, iar tranzacția globală este rolled back.

## Standardul XA (eXtended Architecture)

Standardul XA reprezintă o specificație dezvoltată de consorțiul The Open Group pentru procesarea distribuită de tranzacții. Acest standard descrie interfața dintre un manager de tranzacții global și un manager de tranzacții local.

În cadrul unei tranzacții globale distribuite pot participa resurse de date diferite cum ar fi: baze de date, servere de aplicații, sisteme legacy, cache-uri tranzacționale etc. Standardul XA oferă posibilitatea de a accesa aceste resurse de date distribuite în cadrul aceleiași tranzacții și folosește protocolul 2PC pentru a garanta atomicitatea acestora.

## Scalabilitate pe orizontală/verticală

Scalabilitatea pe orizontală (scale out) înseamnă adăugarea unor noduri noi în cadrul unui sistem (ex: adăugarea de servere web la un cluster web existent).

Scalabilitatea pe verticală (scale up) înseamnă adăugarea unor resurse adiționale nodurilor existente (ex: adăugarea de memorie suplimentară, procesoare etc.).

Toți angajații care câștigă mai mult decât colegii lor de departament

Exemple din curs:

{ a1 | a1 ∈ Angajat ∧ ¬(∃a2)(a2 ∈ Angajat ∧ a1.Dept\_ID = a2.Dept\_ID ∧ a2.Salariu > a1.Salariu) }

sau

{ a1 | a1 ∈ Angajat ∧ (∀a2)(a2 ∈ Angajat ⇒ (a1.Dept\_ID ≠ a2.Dept\_ID ∨ a2.Salariu ≤ a1.Salariu)) }

sau

{ a1 | a1 ∈ Angajat ∧ (∀a2)(a2 ∉ Angajat ∨ a1.Dept\_ID ≠ a2.Dept\_ID ∨ a2.Salariu ≤ a1.Salariu) }

Altă rezolvare propusă de colega voastră:

{ a1 | a1 ∈ Angajat ∧ (∀a2 ∈ Angajat)(a1.Dept\_ID = a2.Dept\_ID ∧ a2.Salariu ≤ a1.Salariu) }

\*\*\* sau expresia echivalentă (folosind regulile logice)

{ a1 | a1 ∈ Angajat ∧ (∀a2)(a2 ∈ Angajat ⇒ (a1.Dept\_ID = a2.Dept\_ID ∧ a2.Salariu ≤ a1.Salariu)) \*\*\* }

Această rezolvare **nu** este corectă deoarece, pentru a se selecta o anumită tuplă a1, expresia (de variabilă *bounded* a2, evaluată în contextul variabilei *free* a1):

( $\forall a2 \in \text{Angajat}$ )( $a1.\text{Dept\_ID} = a2.\text{Dept\_ID} \wedge a2.\text{Salariu} \leq a1.\text{Salariu}$ )

trebuie să se evaluateze pe true. Ca să se evaluateze pe true pentru un anumit a1, înseamnă ca **orice** angajat a2 trebuie să satisfacă expresia ( $a1.\text{Dept\_ID} = a2.\text{Dept\_ID} \wedge a2.\text{Salariu} \leq a1.\text{Salariu}$ ). Adică, **orice** angajat a2 trebuie să fie din același departament cu a1 (partea din expresie:  $a1.\text{Dept\_ID} = a2.\text{Dept\_ID}$ ). Evident, dacă există angajați în cel puțin două departamente, expresia se va evalua **întotdeauna** pe false pentru orice a1, iar rezultatul va fi multimea vidă.