



BAZE DE DATE

**CURS 6**

# ALGEBRA RELAȚIONALĂ

- **LDD** – precizează **entitățile**, relațiile dintre ele, **atributele**, structura atributelor, **cheile**, **constrângerile**, prin urmare definește **structura obiectelor bazei de date** (**schema** bazei de date);
- **LMD** – cuprinde aspecte referitoare la introducerea, modificarea, eliminarea și căutarea datelor;

# ALGEBRA RELAȚIONALĂ

- Modelul relațional oferă **două mulțimi de operatori pe relații**:
  - **algebra relațională** (filtrele se obțin aplicând operatori specializați asupra uneia sau mai multor relații din cadrul bazei relaționale);
  - **calculul relațional** (filtrele se obțin cu ajutorul unor formule logice pe care tuplurile rezultatului trebuie să le satisfacă);
- Echivalența dintre algebra relațională și calculul relațional a fost demonstrată de Jeffrey David Ullman. Această echivalență arată că orice relație posibil de definit în algebra relațională poate fi definită și în cadrul calcului relațional, și reciproc.

# ALGEBRA RELAȚIONALĂ

- **Algebra relationala** a fost introdusa de *Edgar Frank Codd* si are la baza o **multime de operatii** care se aplica asupra unor relatii, rezultatul fiind tot o relatie;
- Este un limbaj abstract cu ajutorul caruia se pot scrie cererile (se poate face o paralela cu pseudocodul care se transforma in cod interpretat de masina);
- Pentru a scrie cereri folosind ***algebra relationala*** avem nevoie de simboluri specifice, operatii si implicit operatori;
- Cu alte cuvinte, in algebra relationala folosim anumiti operatori specifici, reprezentati cu ajutorul unor simboluri si notatii, pe care ii aplicam unor relatii, rezultatul final fiind tot o relatie;

# ALGEBRA RELAȚIONALĂ

**De exemplu:**

Fie  $R$  și  $S$  două **relatii** asupra cărora se aplica operatorul **UNION** (reuniune). Rezultatul reuniunii celor două relatii este tot o relatie și se notează:

$$\text{Rezultat} = \text{UNION}(R, S)$$

Prin algebra relationala se pot construi cererile pas cu pas până la obținerea unui rezultat final, prin scrierea ***expresiilor relationale***.

# ALGEBRA RELAȚIONALĂ

Scopul fundamental al algebrei relaționale este de a permite scrierea **expresiilor relaționale**

- acestea sunt o **reprezentare** de nivel superior, simbolică, a intențiilor utilizatorului și pot fi supuse unei diversități de **reguli de transformare (optimizare)**.

Relațiile sunt închise față de algebra relațională

- operanzii și rezultatele sunt relații → ieșirea unei operații poate deveni intrare pentru alta → posibilitatea imbricării expresiilor în algebra relațională.

# ALGEBRA RELAȚIONALĂ

## Operatorii algebrei relaționale:

- operatori clasici pe mulțimi (**UNION, INTERSECT, PRODUCT, DIFFERENCE**);
- operatori relaționali speciali (**PROJECT, SELECT, JOIN, DIVISION**).

# OPERATORUL PROJECT

- Operatorul **PROJECT** – se utilizeaza atunci cand se doreste preluarea anumitor valori din baza de date (atributele/coloanele  $A_1, A_2, \dots, A_n$ ), fara a fi necesara o conditie.
- Proiecția este o operație unară care elimină anumite atribute ale unei relații producând o submulțime „pe verticală” a acesteia.
  - Suprimarea unor atribute poate avea ca efect apariția unor tupluri duplicate, care trebuie eliminate.
- **Notatii:**
  - **PROJECT** ( $R, A_1, \dots, A_m$ )
  - $\Pi_{A_1, \dots, A_m} (R)$
  - $R[A_1, \dots, A_m]$

unde  $A_1, A_2, \dots, A_m$  sunt parametrii proiecției relativ la relația  $R$ .



# OPERATORUL PROJECT

**Notatie:** *PROJECT(R, A1, A2, ... , An)*

**Exemplu :** Sa se obtina o lista care cuprinde numele si job-ul angajatilor.

**Cerere SQL:**

```
SELECT last_name, job_id  
FROM employees;
```

**Algebra relationala (expresia algebrica):**

**Rezultat** = PROJECT(EMPLOYEES, nume, job);

# OPERATORUL PROJECT

## 1. Proiecție cu dubluri în SQL:

```
SELECT department_id  
FROM employees;
```

## 2. Proiecție fără dubluri în SQL:

```
SELECT DISTINCT department_id  
FROM employees;
```

# OPERATORUL SELECT

- Operatorul **SELECT** – se utilizeaza atunci cand preluam date pe baza unei conditii.
- Selecția (restricția) este o operație unară care produce o submulțime pe „orizontală” a unei relații  $R$ .
  - Această submulțime se obține prin extragerea tuplurilor din  $R$  care **satisfac o condiție specificată**.
- **Notatii:**
  - **SELECT( $R$ , condiție)**
  - $\sigma_{\text{condiție}}(R)$
  - $R[\text{condiție}]$
  - **RESTRICT( $R$ , condiție).**

# OPERATORUL SELECT

**Notatie:** *SELECT(R, conditie)*

**Exemplu :** Sa se obtina toate informatiile despre angajatii care au jobul 'IT\_PROG'.

**Cerere SQL:**

```
SELECT *  
FROM employees  
WHERE job_id = 'IT_PROG';
```

**Algebra relationala (expresia algebrica):**

**Rezultat** = SELECT(EMPLOYEES, job\_id = 'IT\_PROG')

# OPERATORUL UNION

- Operatorul **UNION** – reuniunea a doua relații  $R$  și  $S$  (elementele comune și necomune afișate o singură dată).
- **Reuniunea** a două relații  $R$  și  $S$  este mulțimea tuplurilor aparținând fie lui  $R$ , fie lui  $S$ , fie ambelor relații.
- **Notății:**
  - **UNION( $R, S$ )**
  - $R \cup S$
  - $OR(R, S)$
  - $APPEND(R, S)$

# OPERATORUL UNION

**Notatie:** *UNION(R, S)*

**Exemplu:** Se cer codurile departamentelor al caror nume contine sirul “re” sau in care lucreaza angajati avand codul job-ului “SA\_REP”.

**Cerere SQL:**

```
SELECT department_id  
FROM departments  
WHERE lower(department_name) like '%re%'
```

UNION

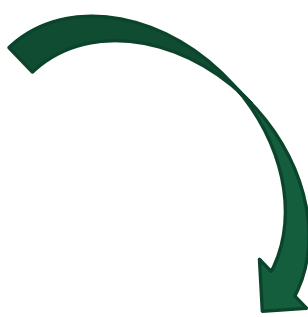
```
SELECT department_id  
FROM employees  
WHERE upper(job_id) like '%SA_REP%';
```

# OPERATORUL UNION

```
SELECT department_id
FROM departments
WHERE lower(department_name) like '%re%'

UNION

SELECT department_id
FROM employees
WHERE upper(job_id) like '%SA_REP%';
```



## Algebra relationala (expresia algebrica):

*R1 = SELECT(DEPARTMENTS, department\_id LIKE '%re%')*

*R2 = PROJECT(R1, department\_id)*

*R3 = SELECT(EMPLOYEES, job\_id LIKE '%SA\_REP%')*

*R4 = PROJECT(R3, department\_id)*

**Rezultat** = *UNION(R2, R4)*

# OPERATORUL UNION

**Algebra relationala (expresia algebrica):**

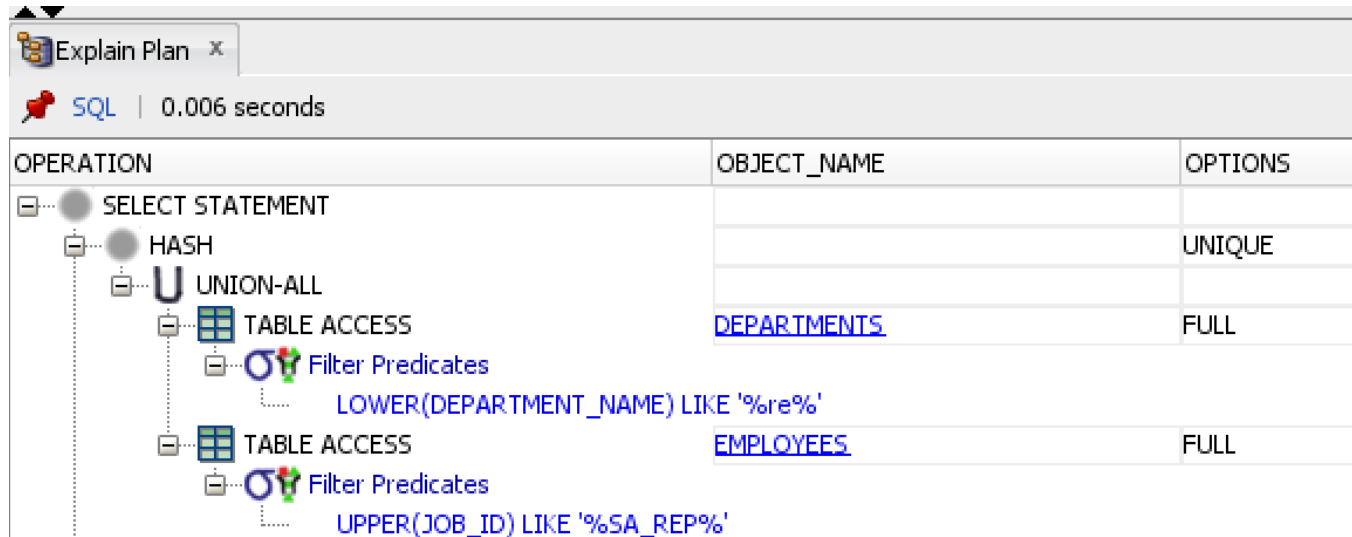
*R1 = SELECT(DEPARTMENTS, department\_id LIKE '%re%')*

*R2 = PROJECT(R1, department\_id)*

*R3 = SELECT(EMPLOYEES, job\_id LIKE '%SA\_REP%')*

*R4 = PROJECT(R3, department\_id)*

**Rezultat** = *UNION(R2, R4)*





# OPERATORUL DIFFERENCE

- Operatorul **DIFFERENCE** – diferența a două relații  $R$  și  $S$  (elementele care sunt în  $R$  și nu sunt în  $S$ ) – este mulțimea tuplurilor care aparțin lui  $R$ , dar nu aparțin lui  $S$ .
- Diferența este o operație binară necomutativă care permite obținerea tuplurilor ce apar numai într-o relație.
- **Notatii:**
  - **DIFFERENCE( $R, S$ )**
  - $R - S$
  - **REMOVE( $R, S$ )**
  - **MINUS( $R, S$ )**

# OPERATORUL DIFFERENCE

Notatie: *DIFFERENCE(R, S)*

**Exemplu:** Sa se obtina codurile departamentelor in care nu lucreaza nimeni.

**Cerere SQL:**

```
SELECT department_id
FROM departments -- department_id este cheie primara, deci avem o lista
                  -- unica de departamente
```

**MINUS** -- din lista tuturor departamentelor dorim sa eliminam  
-- departamentele in care lucreaza angajati

```
SELECT department_id
FROM employees; -- department_id este cheie externa, deci sunt
                 -- departamente in care lucreaza angajati
```

**Algebra relationala (expresia algebrica):**

R1 = PROJECT(DEPARTMENTS, department\_id)

R2 = PROJECT(EMPLOYEES, department\_id)

**Rezultat** = DIFFERENCE(R1, R2)

# OPERATORUL INTERSECT

- Operatorul **INTERSECT** – intersectia a doua relatii  $R$  si  $S$  (elementele comune celor doua relatii afisate o singura data).
- Intersecția a două relații  $R$  și  $S$  este mulțimea tuplurilor care aparțin atât lui  $R$ , cât și lui  $S$ . Operatorul INTERSECT este un operator binar, comutativ, derivat:
  - $R \cap S = R - (R - S)$
  - $R \cap S = S - (S - R)$ .
- **Notatii:**
  - **INTERSECT(R, S)**
  - $R \cap S$
  - **AND(R, S)**
- Operatorii **INTERSECT** și **DIFFERENCE** pot fi simulați în SQL (în cadrul comenzii SELECT) cu ajutorul opțiunilor EXISTS, NOT EXISTS, IN, NOT IN ( != ANY) – o sa implementam in laborator toate exemplele

# OPERATORUL INTERSECT

**Notatie:** *INTERSECT(R, S)*

**Exemplu:** Sa se afiseze angajatii (codul lor) care sunt manageri in cadrul departamentelor si in acelasi timp sunt si managerii altor angajati.

**Cerere SQL:**

```
SELECT manager_id  
FROM departments -- managerii de departament
```

**INTERSECT**

```
SELECT manager_id  
FROM employees; -- managerii angajatilor
```

**Algebra relationala (expresia algebrica):**

$R1 = \text{PROJECT}(\text{DEPARTMENTS}, \text{manager\_id})$

$R2 = \text{PROJECT}(\text{EMPLOYEES}, \text{manager\_id})$

**Rezultat** =  $\text{INTERSECT}(R1, R2)$

# OPERATORUL INTERSECT

Intersectia se poate simula utilizand operatorul **IN** sau operatorul **EXISTS** (il vom studia in laborator).

Cum procedam atunci cand utilizam operatorul **IN**?

**Cerere SQL** (simularea intersectiei folosind operatorul **IN**):

```
SELECT manager_id  
FROM departments  
WHERE manager_id IN (SELECT manager_id  
                      FROM employees );
```

# OPERATORUL PRODUCT

- Operatorul **PRODUCT** – produsul cartezian dintre relatia  $R$  si relatia  $S$ .
- Fie  $R$  și  $S$  relații de aritate  $m$ , respectiv  $n$ . Produsul cartezian al lui  $R$  cu  $S$  este mulțimea tuplurilor de aritate  $m + n$  unde primele  $m$  componente formează un tuplu în  $R$ , iar ultimele  $n$  componente formează un tuplu în  $S$ .
- **Notatii:**
  - **PRODUCT( $R, S$ )**
  - $R \times S$
  - **TIMES( $R, S$ )**

# OPERATORUL PRODUCT

**Notatie:** *PRODUCT(R, S)*

**Exemplu:** Sa se afiseze toti angajatii (codurile si numele angajatilor) impreuna cu toate departamentele (numele si codurile departamentelor).

**Cerere SQL:**

```
SELECT employee_id, last_name, d.department_id, department_name  
FROM employees e, departments d;
```

**Algebra relationala (expresia algebrica):**

```
R1 = PROJECT(EMPLOYEES, employee_id, last_name)  
R2 = PROJECT(DEPARTMENTS, department_id, department_name)  
Rezultat = PRODUCT(R1, R2)
```

**Cerere SQL (utilizand CROSS JOIN):**

```
SELECT employee_id, last_name, d.department_id, department_name  
FROM employees e CROSS JOIN departments d;
```

# OPERATORUL JOIN

- Operatorul **JOIN** – permite regasirea informatiei din mai multe relatii corelate. Operatorul combina **produsul cartezian** cu **selectia** si **proiectia**.
- 4 tipuri de **JOIN**:
  - **NATURAL JOIN**
  - **θ-JOIN**
  - **SEMI-JOIN**
  - **OUTER JOIN**



# NATURAL JOIN

- **NATURAL JOIN** – operatorul de compunere naturală combină tupluri din două relații  $R$  și  $S$ , cu condiția ca **atributele comune să aibă valori identice**.
- Cu alte cuvinte, acest tip de join realizează join automat bazat pe coloanele comune existente în cele două tabele între care se realizează operația de join. Coloanele comune sunt cele care au același nume în ambele tabele.

# NATURAL JOIN

Algoritmul care realizează compunerea naturală este următorul:

1. se calculează produsul cartezian  $R \times S$ ;
2. pentru fiecare atribut comun  $A$  care definește o coloană în  $R$  și o coloană în  $S$ , se selectează din  $R \times S$  tuplurile ale căror valori coincid în coloanele  $R.A$  și  $S.A$  (atributul  $R.A$  reprezintă numele coloanei din  $R \times S$  corespunzătoare coloanei  $A$  din  $R$ );
3. pentru fiecare astfel de atribut  $A$  se elimină coloana  $S.A$ , iar coloana  $R.A$  se va numi  $A$ .

# NATURAL JOIN

- Operatorul NATURAL JOIN poate fi exprimat formal astfel:

$$\text{JOIN}(R, S) = \Pi_{i_1, \dots, i_m} \sigma_{(R.A_1 = S.A_1) \wedge \dots \wedge (R.A_k = S.A_k)}(R \times S),$$

unde  $A_1, \dots, A_k$  sunt attributele comune lui  $R$  și  $S$ , iar  $i_1, \dots, i_m$  reprezintă lista componentelor din  $R \times S$  (păstrând ordinea inițială) din care au fost eliminate componentele  $S.A_1, \dots, S.A_k$ .

# NATURAL JOIN

Notatie: *JOIN(R, S)*

**Exemplu:** Sa se afiseze informatii despre angajatii care lucreaza in departamente si care sunt in acelasi timp atat manageri de departament, cat si managerii altor angajati.

**Algoritmul compunerii naturale:**

1. Se calculeaza produsul cartezian dintre relatiile R si S (R reprezinta relatia employees, iar S relatia departments);

```
SELECT employee_id, last_name, d.department_id,  
department_name, d.manager_id  
FROM employees e, departments d;
```

# NATURAL JOIN

2. Se selecteaza tuplurile pentru care exista valori comune atat in relatia R, cat si in relatia S (valorile comune sunt – department\_id si manager\_id – acestea facand parte din ambele relatii)

*WHERE e.department\_id = d.department\_id AND  
e.manager\_id = d.manager\_id;*

3. Pentru fiecare atribut existent in ambele relatii, in afisare se pastreaza doar o singura data coloana comuna (in cazul nostru o sa avem o singura data coloana department\_id si o singura data coloana manager\_id).

# NATURAL JOIN

Algoritmul anterior se aplica automat atunci cand utilizam **NATURAL JOIN**, nefiind nevoie de aliasuri.

```
SELECT employee_id, last_name, department_id, department_name,  
manager_id  
FROM employees NATURAL JOIN departments;
```

**Algebra relationala (expresia algebrica):**

**Rezultat** = JOIN(EMPLOYEES, DEPARTMENTS)

# $\theta$ -JOIN

- **$\theta$ -JOIN** – combina tupluri (coloane/atribute) din doua relatii R si S pe baza unei conditii specificata in cadrul operatiei de JOIN. Aceasta conditie poate fi bazata pe egalitatea unor coloane (de obicei cheia primara si cheia externa) – acesta fiind un caz particular de  **$\theta$ -JOIN**, si anume **EQUIJOIN (sau inner join)**, dar sunt utilizate in egala masura si conditii care implica operatorii <, <=, >, >=, != (acest tip de join fiind un **NONEQUIJOIN**)
- Operatorul  **$\theta$ -JOIN** este un operator derivat, fiind o combinație de produs scalar și selecție:

$$\text{JOIN}(R, S, \text{condiție}) = \sigma_{\text{condiție}} (R \times S)$$

# $\theta$ -JOIN

**Notatie:** *JOIN(R, S, conditie)*

**Exemplu:** Sa se afiseze numele angajatilor impreuna cu denumirea jobului in cadrul caruia lucreaza.

**Cerere SQL:**

```
SELECT last_name, job_title  
FROM employees e, jobs j  
WHERE e.job_id = j.job_id;
```

**Algebra relationala (expresia algebrica):**

$R1 = \text{PROJECT}(\text{EMPLOYEES}, \text{last\_name})$

$R2 = \text{PROJECT}(\text{JOBS}, \text{job\_title})$

**Rezultat** =  $\text{JOIN}(R1, R2, \text{EMPLOYEES.job\_id} = \text{JOBS.job\_id})$



# SEMI-JOIN

- **SEMI-JOIN** – se utilizeaza atunci cand din cele doua relatii participante *R* si *S* se utilizeaza doar attributele relatiei *R* sau doar cele ale relatiei *S*.
- Operatorul SEMI-JOIN conservă **attributele unei singure relații participante** la compunere și este utilizat când nu sunt necesare toate attributele compunerii. Operatorul este asimetric.
  - Tupluri ale relației *R* care participă în compunerea (naturală sau  $\theta$ -JOIN) dintre relațiile *R* și *S*.
- SEMI-JOIN este un operator derivat, fiind o combinație de proiecție și compunere naturală sau proiecție și  $\theta$ -JOIN:

Notatie:  $SEMIJOIN(R, S) = PROJECT(JOIN(R, S))$   
 $SEMIJOIN(R, S, conditie) = PROJECT(JOIN(R, S, conditie))$

# SEMI-JOIN

**Exemplu:** Sa se afiseze departamentele (codul si denumirea) care au cel putin un angajat.

Se afiseaza in final coloane care fac parte doar dintr-un tabel – departments => operatorul SEMI-JOIN conservă **atributele unei singure relații participante**.

```
SELECT d.department_id, department_name  
FROM employees e, departments d  
WHERE e.department_id = d.department_id;
```

**Algebra relationala (expresia algebrica):**

```
R1 = JOIN(EMPLOYEES, DEPARTMENTS, EMPLOYEES.department_id =  
DEPARTMENTS.department_id)  
Rezultat = PROJECT(R1, department_id, department_name)
```

# OUTER JOIN

- **OUTER JOIN** – pentru doua relatii  $R$  si  $S$ , acest operator returneaza valorile atributelor care indeplinesc conditia de corelare (adica intersectia celor doua relatii pe baza unei valori comune – join clasic), impreuna cu valori NULL in functie de tipul acestui operator.
- Cu alte cuvinte, operația de **compunere externă** combină tupluri din două relații, tupluri pentru care nu sunt satisfăcute condițiile de corelare.
- În cazul aplicării operatorului JOIN se pot pierde tupluri, atunci când **există un tuplu în una din relații pentru care nu există nici un tuplu în cealaltă relație**, astfel încât să fie satisfăcută relația de corelare.
- Operatorul elimină acest inconvenient prin **atribuirea valorii null** valorilor atributelor care există într-un tuplu din una dintre relațiile de intrare, dar care nu există și în cea de-a doua relație.
- Practic, se realizează compunerea a două relații  $R$  și  $S$  la care se adaugă tupluri din  $R$  și  $S$ , care nu sunt conținute în compunere, completate cu valori *null* pentru attributele care lipsesc.

# OUTER JOIN

➤ Compunerea externă poate fi: LEFT, RIGHT, FULL.

➤ De exemplu:

**R LEFT OUTER JOIN S** – reprezintă compunerea în care tuplurile din *R*, care nu au valori similare în coloanele comune cu relația *S*, sunt de asemenea incluse în relația rezultat.

➤ Asadar,

**R LEFT OUTER JOIN S** – se vor afisa valorile din *R* si *S* care respecta conditia de JOIN (intersectia) impreuna cu valorile care sunt in *R* (LEFT) si nu sunt in *S*. In acest caz valorile care lipsesc vor fi automat completate cu NULL.

# OUTER JOIN

## Exemplu:

Consideram tabelul **EMPLOYEES** ca fiind relatia **R** si tabelul **DEPARTMENTS** ca fiind relatia **S**

Sa se afiseze angajatii (cod si nume) impreuna cu departamentele in care lucreaza (cod si denumire). Rezultatul o sa includa toti angajatii chiar daca nu au departament.

```
SELECT employee_id, d.department_id, last_name, department_name  
FROM employees e LEFT JOIN departments d ON (e.department_id =  
d.department_id);
```

# OUTER JOIN

- In output se observa angajatii care lucreaza intr-un departament, adica valorile din **EMPLOYEES** si **DEPARTMENTS** (din *R* si *S*) care indeplinesc conditia de corelare (**e.department\_id = d.department\_id**), impreuna cu valorile care sunt in **EMPLOYEES** (*R*) si nu sunt in **DEPARTMENTS** (*S*), adica angajatii care nu au departament.
- Valorile care lipsesc vor fi completate automat cu **NULL** – in exemplul nostru valoarea care lipseste este cea specifica departamentului deoarece sunt afisati si angajatii **care nu au departament**, ceea ce inseamna ca angajatul exista, dar departamentul nu. In acest caz o sa existe un rezultat ca acesta :

**178   NULL   Grant   NULL** – coloanele reprezentative departamentului (department\_id si department\_name sunt automat completate cu valoarea NULL)

# OUTER JOIN

- In mod asemanator se vor comporta si variantele urmatoare de **JOIN** (*RIGHT OUTER JOIN* si *FULL OUTER JOIN*)
- **R RIGHT OUTER JOIN S** – se vor afisa valorile din R si S care respecta conditia de JOIN (intersectia) impreuna cu valorile care sunt in S (RIGHT) si nu sunt in R. In acest caz valorile care lipsesc vor fi automat completate cu NULL.
- **R FULL OUTER JOIN S** – se vor afisa valorile din R si S care respecta conditia de JOIN (intersectia) impreuna cu valorile care sunt in R si nu sunt in S si valorile care sunt in S si nu sunt in R. In acest caz valorile care lipsesc vor fi automat completate cu NULL.

# OPERATORUL DIVISION

- **Diviziunea** este o operație binară care definește o relație ce conține valorile atributelor dintr-o relație care apar **în toate** valorile atributelor din cealaltă relație.
- **Notatii:**
  - **DIVISION(R, S)**
  - **DIVIDE(R, S)**
  - **$R \div S$**
- Diviziunea conține acele tupluri de dimensiune  $n - m$  la care, adăugând orice tuplu din  $S$ , se obține un tuplu din  $R$ .
- Operatorul diviziune poate fi exprimat formal astfel:
  - **$R^{(n)} \div S^{(m)} = \{t^{(n-m)} \mid \forall s \in S, (t, s) \in R\}$** , unde  $n > m$  și  $S \neq \emptyset$ .



# ALGEBRA RELAȚIONALĂ

- Operatorul **DIVISION** este legat de cuantificatorul universal ( $\forall$ ) care nu există în SQL.

- Cuantificatorul universal poate fi însă simulat cu ajutorul cuantificatorului existențial ( $\exists$ ) utilizând relația:

$$\forall x P(x) \equiv \neg \exists x \neg P(x).$$

- Prin urmare, operatorul **DIVISION** poate fi exprimat în SQL prin succesiunea a doi operatori NOT EXISTS.

- **Acest operator o sa fie studiat in cadrul laboratorului**