Components lògics d'una base de dades

- **Objectius**

  – Completar els components lògics d'una base de dades vistos a les sessions de laboratori

  – Conèixer les sentències que ofereix el llenguatge estàndard SQL

  – Poder fer servir les sentències que ofereix un sistema relacional concret, coneixent les diferències respecte al llenguatge estàndard

Components lògics d'una base de dades

- **Temari**

  Components Lògics de Dades

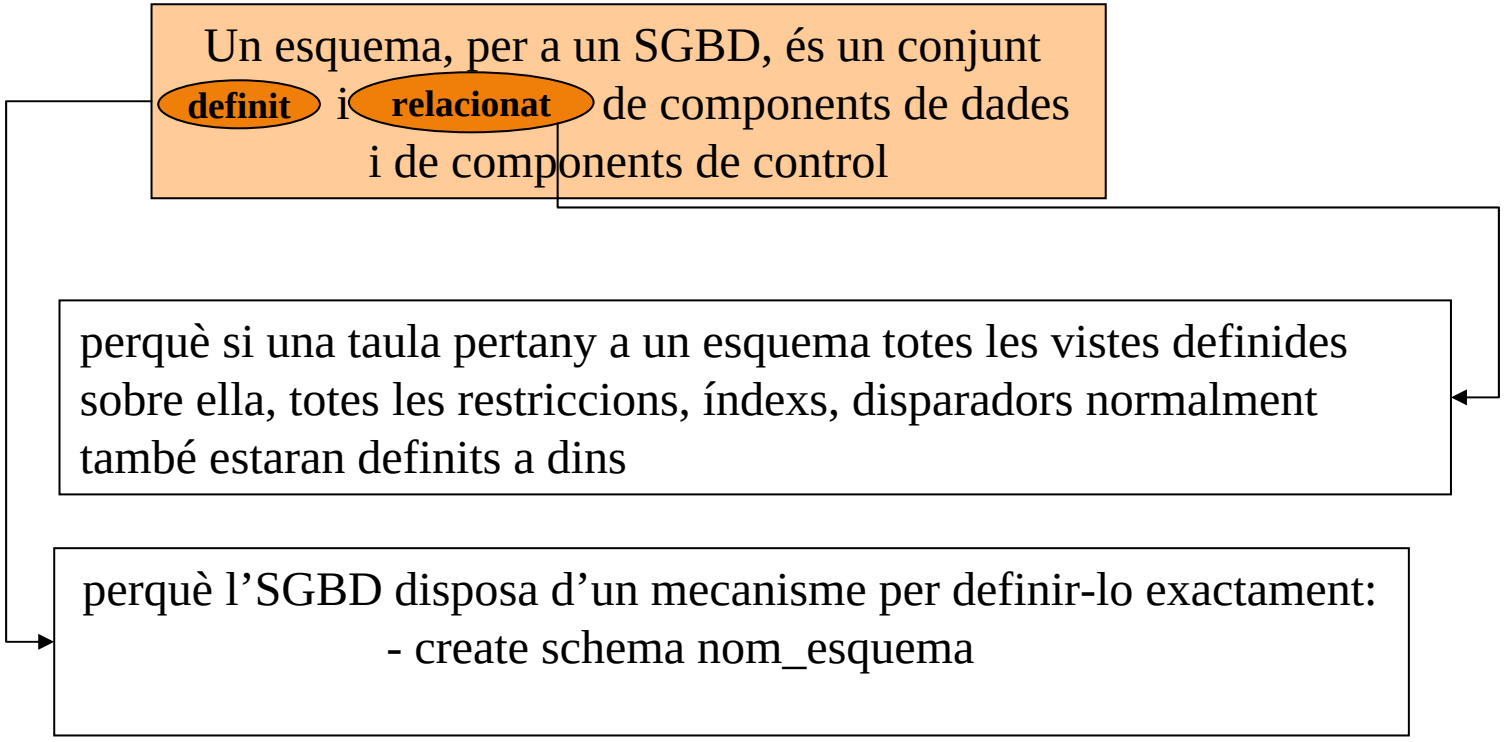
  - Esquemes
  - Dominis i Taules
  - Assercions
  - Vistes

  Components Lògics de Control

  - Procediments
  - Disparadors
  - Privilegis

Esquemes

- Els SGBD agrupen els components de dades (taules, vistes, …) i els components de control (procediments, disparadors, …), que veurem més endavant, en un altre component lògic anomenat *esquema*



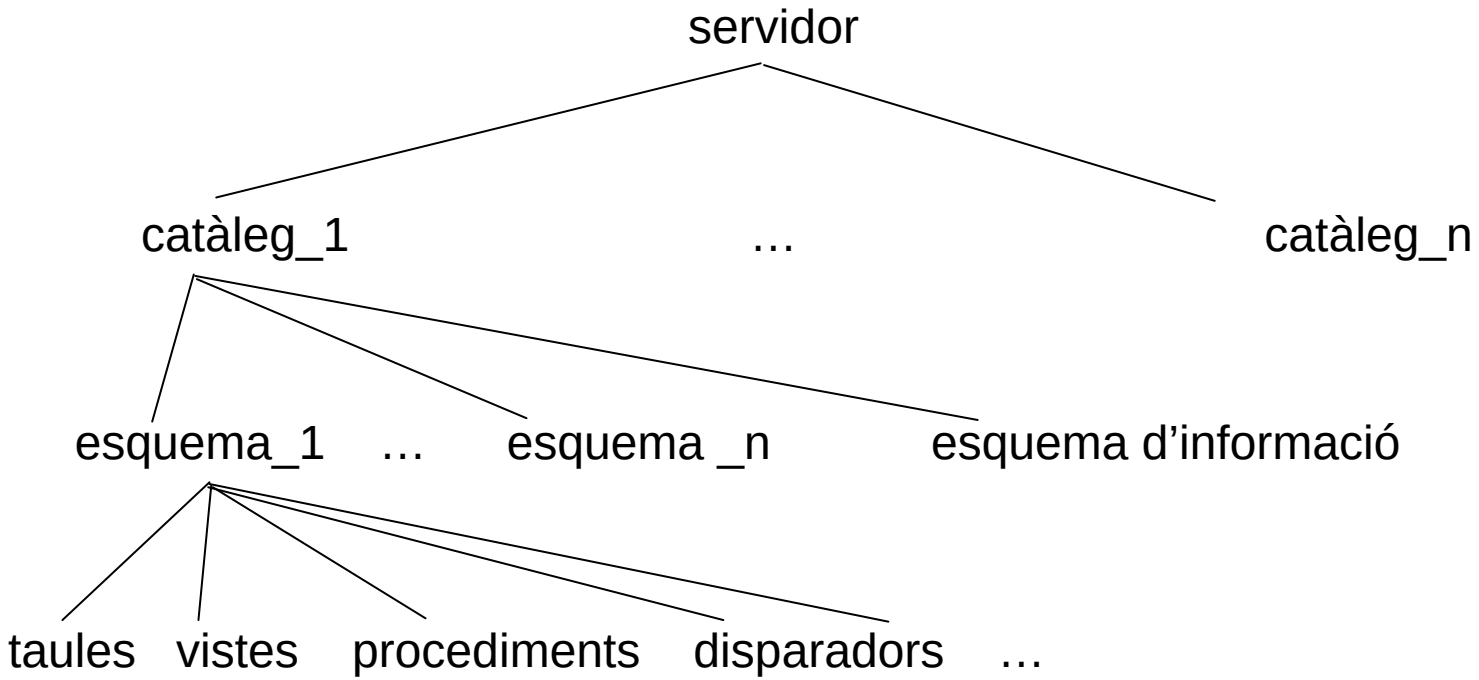
Esquemes

- Des d'aquest punt de vista, el component esquema és una eina especifica de l'SGBD que serveix com a unitat administrativa per agrupar un conjunt d'altres components

- Aquesta és la utilitat principal del component esquema: ens permet centralitzar tasques administratives que d'una altra manera hauríem de repetir individualment. Podem engegar, aturar, o atorgar privilegis d'un conjunt de components lògics a un usuari

- Hi ha diversos criteris per a decidir quins components constituiran un esquema. Per exemple:
  - Per aplicacions
  - Per usuaris finals

Servidor, Catàleg i Esquema



- Un catàleg (catalog) és un grup d'esquemes, un dels quals es anomenat esquema d'informació (information schema)
- L'esquema d'informació és un conjunt de vistes que conté la descripció de totes les dades SQL que pertanyen al catàleg corresponent
- Un servidor (cluster) pot contenir zero o més catàlegs

Esquemes

- No hi ha una instrucció estàndar per crear, destruir o modificar Catàlegs. Dependrà de cada implementador

- La sentència **CREATE SCHEMA** dona nom a un nou esquema, identifica al propietari de l'esquema i pot donar la llista d'elements de l'esquema

CREATE SCHEMA [[nom_cat.]nom_esq] **AUTHORIZATION** ident_usuari
[llista d'elements de l'esquema];

- La sentència **DROP SCHEMA** amb l'opció **RESTRICT** esborra un esquema només si aquest està completament buit. En canvi amb l'opció **CASCADE** l'esborra encara que contingui elements

DROP SCHEMA nom_esquema [**RESTRICT | CASCADE**];

Connexions, Sessions i Transaccions

- Una **connexió** és una associació que es crea entre un client SQL i un servidor SQL:

  CONNECT TO nom_servidor [**AS** nom_connexio] [**USER** ident_usuari];
  SET SCHEMA nom_esq;

  I que es destrueix quan acaba:

  DISCONNECT nom_connexio | **DEFAULT** | **CURRENT** | **ALL**;

- Una **sessió** és el context en el que un usuari o aplicació executa una seqüència de sentències SQL mitjançant una connexió:

  SET SESSION CHARACTERISTICS AS mode_transac [, mode_transac ...];

Connexions, Sessions i Transaccions

- Una **transacció** és una seqüència de sentències SQL atòmica amb respecte a la recuperació que s'inicia amb moltes sentències SQL, com ara create schema, select, insert, delete o update. Les seves característiques es poden definir prèviament amb:

  SET TRANSACTION mode_transac [, mode_transac ...];

  on mode_transac pot ser: mode_d'accés | nivell d'aïllament

  on mode_d'accés pot ser: **READ ONLY** | **READ WRITE**

  on nivell d'aïllament pot ser: **READ UNCOMMITTED |**
  READ COMMITTED | REPETEABLE READ | SERIALIZABLE

  És pot fer explícit l'inici d'una transacció amb:

  START TRANSACTION mode_transac [, mode_transac ...];

- Una **transacció** finalitza amb les sentències:

  COMMIT [WORK] [AND [NO] CHAIN];

  ROLLBACK [WORK] [AND [NO] CHAIN];

Dominis

- Un esquema pot contenir zero o més dominis. Un domini és un conjunt de valors vàlids definits per l'usuari

CREATE DOMAIN nom_domini **AS** tipus_dades
[DEFAULT literal | temps | **USER** | **NULL**]
[llista_restriccions_domini];

on llista_restriccions_domini poden ser:

CONSTRAINT nom_restricció
CHECK condició [, **CONSTRAINT** n_r **CHECK** cond ...]

```
CREATE DOMAIN ciutat AS char(15)
        DEFAULT 'BCN'
        CONSTRAINT vàlides CHECK VALUE IN ('BCN', 'MAD');

CREATE TABLE empleats (nempl integer, ciutat_naix ciutat,
    ciutat_resid ciutat, ciutat_treb ciutat);
```

Taules

- El component lògic principal d'una base de dades és la **taula**

CREATE TABLE nom_taula
(definició_columna [, definició_columna ...][, restriccions_taula]);

on definició_columna pot ser:
nom_columna tipus_dades [def_defecte] [restricció_columna]

on def_defecte (definicions per defecte) pot ser:
DEFAULT literal | funció | **NULL**

on funció pot ser:
- funcions de valors de temps: **CURRENT_DATE**, **CURRENT_TIME**, **CURRENT_TIMESTAMP**, ...
- funcions de valors d'usuaris: **CURRENT_USER**, **SESSION_USER**, **SYSTEM_USER** ...

Taules

on tipus_dades pot ser:

- DATE
- TIME
- TIMESTAMP
- INTERVAL
- BINARY LARGE OBJECT(BLOB)
- DECIMAL (NUMERIC)
- FLOAT
- DOUBLE PRECISION
- REAL
- INTEGER (INT)
- SMALLINT
- CHARACTER (CHAR)
- CHARACTER LARGE OBJECT(CLOB)
- VARCHAR
- BOOLEAN

Restriccions de columna

on restricció_columna pot ser:

- **NOT NULL**
- **UNIQUE**
- **PRIMARY KEY**
- **REFERENCES** taula [(columna)]
  [**ON DELETE**
  NO ACTION | RESTRICT | CASCADE | SET NULL | SET DEFAULT]
  [**ON UPDATE**
  NO ACTION | RESTRICT | CASCADE | SET NULL | SET DEFAULT]
- **CHECK** (condicions)
  Les condicions són típicament de rang del domini, encara que permeten posar qualsevol expressió. No usen altres columnes, ni columnes d'altres taules
  Podem donar nom a les restriccions de columna:
  [CONSTRAINT nom_restricció]

Restriccions de columna

- **1er Problema** (Ramakrishnan & Gehrke 98, pàg 217)
  - No es viola mai si la relació està buida

```
CREATE TABLE  emp ( nemp integer,
                    CHECK ((SELECT COUNT(*) FROM emp) > 0)
```

- **2on Problema** (Ullman & Widom 99, pàg 341)
  - Només es comproven quan s'actualitza la taula on estan definides

```
CREATE TABLE  emp
      ( nemp integer,
        dept  char(10) CHECK ( dept IN SELECT dept FROM dept))
```

INSERT, UPDATE empleat **OK** ; DELETE, UPDATE dept **NO**

Restriccions de taula

on restriccions_taula poden ser:

- **UNIQUE** (columna, ...)
- **PRIMARY KEY**(columna,....)
- **FOREIGN KEY**(columna,...) **REFERENCES** taula (columna,...)
  [**ON DELETE**
  NO ACTION | RESTRICT | CASCADE | SET NULL | SET DEFAULT]
  [**ON UPDATE**
  NO ACTION | RESTRICT | CASCADE | SET NULL | SET DEFAULT]
- **CHECK** (condicions)
  No s'usen columnes d'altres taules
  Podem donar nom a les restriccions de taula:
  [CONSTRAINT nom_restricció]

Taules

Exemple:

```
CREATE TABLE persona (
    Dni char(8) PRIMARY KEY,
    Nom varchar(30) NOT NULL,
    Data_naixement date NOT NULL
     CONSTRAINT data_naix CHECK (data_naixement>='01/01/1900'),
    Data_defuncio date,
    Ciutat_naixement varchar(30) NOT NULL,
    Ciutat_residencia varchar(30) NOT NULL,
    Estudis char(1) DEFAULT '1'
     CONSTRAINT estudis CHECK (estudis BETWEEN '1' and '5'),
    Telefon decimal(9) UNIQUE,
     CONSTRAINT dates CHECK ((data_naixement<data_defuncio) OR
            (data_defuncio is NULL)));
```

Assercions

- Restriccions d´integritat que afecten a més d'una taula
- A diferència de les restriccions de columna o de taula es comproven sempre
- En la majoria dels sistemes actuals no es poden definir. Cal usar altres mecanismes: disparadors
- **CREATE ASSERTION** nom **CHECK** (condició);
- Exemple:

empleat(nemp, ciutat_e, ndept) dept(ndept, ciutat_d)

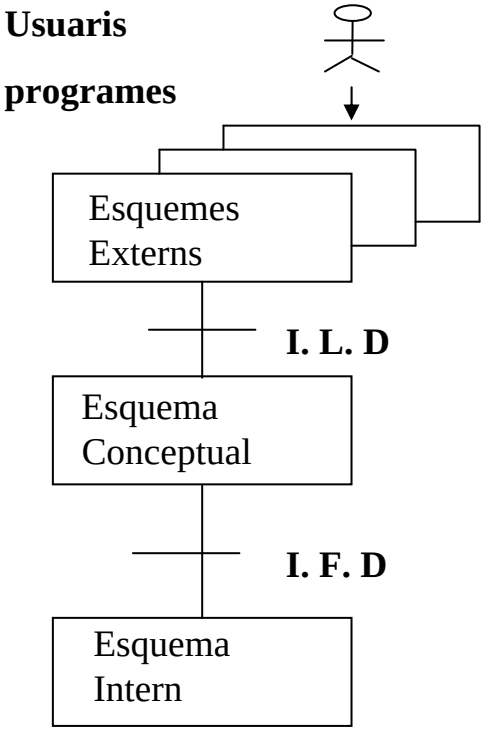
Cal assegurar que tots els empleats treballin a un departament que estigui situat a la ciutat on resideixen!

```
CREATE ASSERTION ciutat_emp_dept CHECK
    (NOT EXISTS (SELECT *
                 FROM empleat e, dept d
                 WHERE e.ndept = d.ndept and
                       e.ciutat_e <> d.ciutat.d));
```

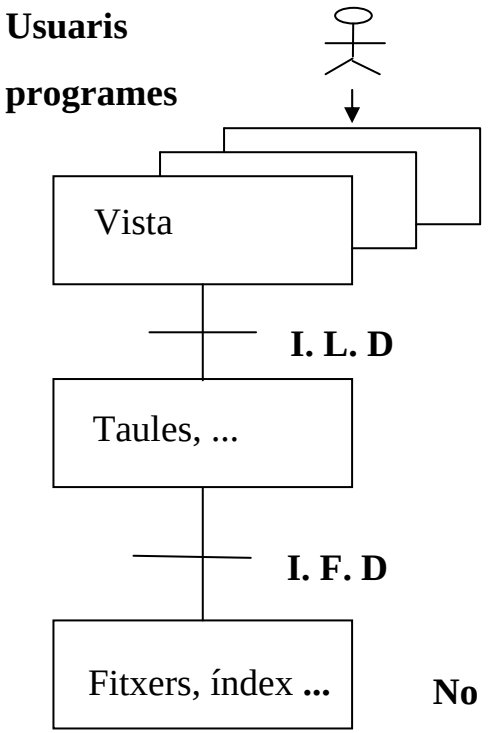
Vistes

Visió general

Arquitectura ANSI/SPARC



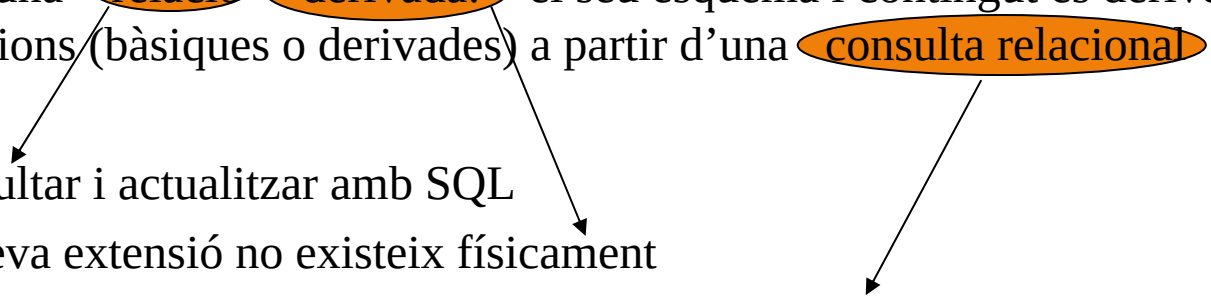
Arquitectura Relacional



No estàndard

Vistes

- Una vista és una relació derivada: el seu esquema i contingut es deriven d'altres relacions (bàsiques o derivades) a partir d'una consulta relacional



Es pot consultar i actualitzar amb SQL

La seva extensió no existeix físicament

Potència SQL per definir vistes, excepte ORDER BY

CREATE VIEW nom_vista [(nom_columna, …)] **AS** sentència_select [**WITH CHECK OPTION**];

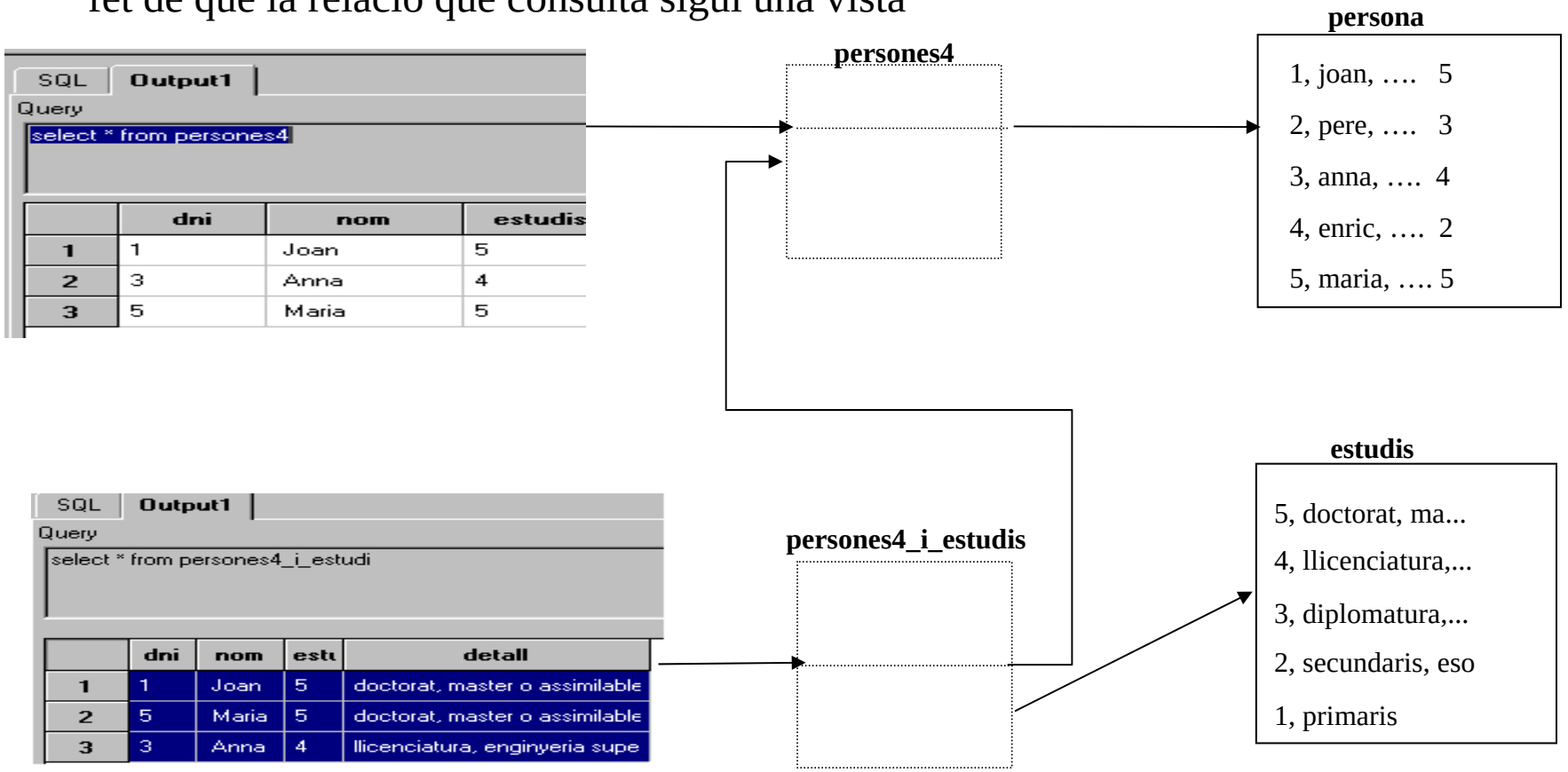
Exemple:

persona (dni, nom, data_naixement, data_defuncio, ciutat_naixement, ciutat_residencia, estudis, telefon)

```
CREATE VIEW persones4 AS  SELECT dni, nom, estudis
                          FROM persona
                          WHERE estudis >= 4
```

Vistes

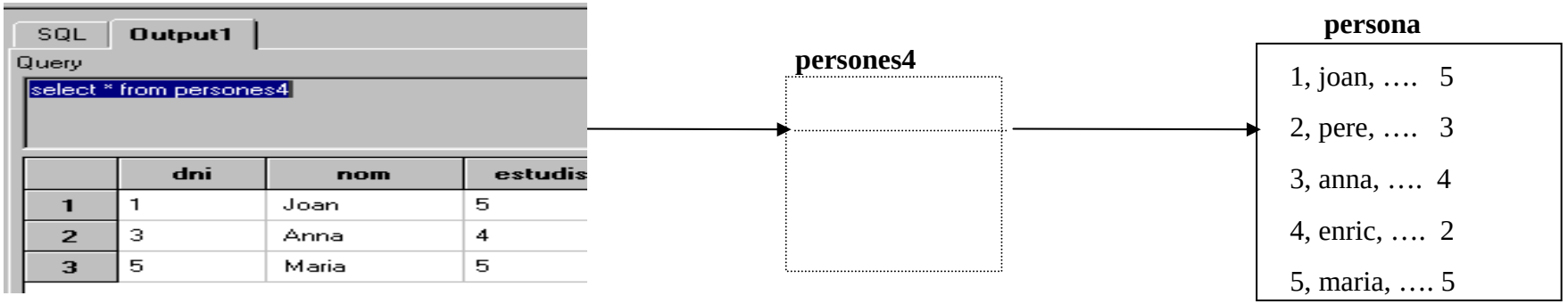
- Les vistes són consultables amb SQL. De fet, de cara a l'usuari final és transparent el fet de que la relació que consulta sigui una vista




```
CREATE VIEW persones4_i_estudis AS SELECT dni, nom, estudis.estudis, estudis.detall
                                   FROM persones4, estudis
                                   WHERE estudis.estudis = persones4.estudis;
```

Vistes

- Les vistes són actualizables amb SQL. De fet, de cara al usuari final és transparent el fet que la relació que actualitzi sigui una vista



	dni	nom	estudis
1	1	Joan	5
2	3	Anna	4
3	5	Maria	5

1. UPDATE persones4 SET nom=’Anna Maria' WHERE dni=5;
2. SELECT * FROM persones4;

	dni	nom	estu
1	1	Joan	5
2	3	Anna	4
3	5	Anna Maria	5

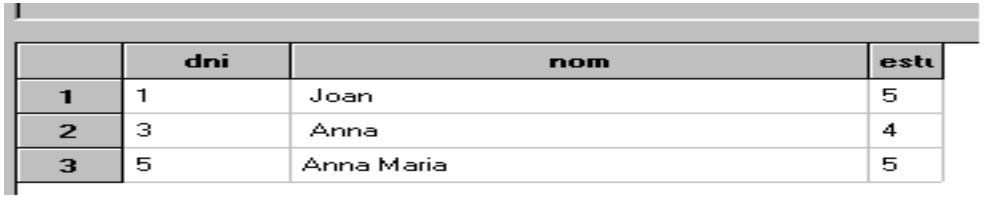
3. UPDATE persones4 SET estudis=2 WHERE dni = 5;
4. SELECT * FROM persones4;

	dni	nom	estu
1	1	Joan	5
2	3	Anna	4

Vistes

```
CREATE VIEW persones4 AS  SELECT dni, nom, estudis
                          FROM persona
                          WHERE estudis >= 4
                          WITH CHECK OPTION;
```

1. UPDATE persones4 SET nom=’Anna Maria' WHERE dni=5;
2. SELECT * FROM persones4;



	dni	nom	estu
1	1	Joan	5
2	3	Anna	4
3	5	Anna Maria	5

3. UPDATE persones4 SET estudis=2 WHERE dni = 5;

Error!

Vistes

• De totes maneres, no totes les vistes són actualitzables. L'estàndard defineix amb precisió quines ho són i quines no. De manera simplificada, segons SQL-92, s'admeten només actualizacions d'aquelles vistes definides com:

  • Operació de selecció de l'àlgebra relacional sobre una única relació (o vista actualitzable)

  • Operació de selecció més projecció de l'àlgebra relacional, sempre i quan la projecció inclogui els atributs clau i els atributs amb restricció no null.

• Per tant: no subselect, no joins, no agregats ...

• El motiu de fons és que amb aquest tipus de vista qualsevol actualització de la vista sempre es pot traduir a una única actualització de la taula de base i, per tant, sense ambigüitat.

Vistes

subministraments(nprov, nmat, qtt)

p1	m1	100
p2	m1	200
p2	m2	200

```
CREATE VIEW sumaqtt(material,suma) AS SELECT nmat, sum(qtt)
                                   FROM subministraments
                                   GROUP BY nmat
```

1. DELETE de “m1,300” de la vista => DELETE “p1, m1, 100”
   DELETE “p2, m1, 200”

   Una única solució! Hauria de ser possible traduir-la.

   Però no ho és ni a l’estàndard, ni als SGBD concrets

2. UPDATE de “m1, 300” per “m1, 301” =>

   Com es tradueix? Hi ha múltiples solucions !!!

Vistes

proveidors(nprov, nom, ...)
100 joan

subministraments(nprov, nmat, qtt,...)
100 m1 5230

```
CREATE VIEW prov_subprov AS SELECT *
                            FROM proveidors, subministraments
                            WHERE proveidors.nprov = subministraments.nprov
```

1. DELETE "100, joan,...m1,..." de la vista =>
   - 1.1 DELETE "100, m1, 5230, ..." de subministraments
   - 1.2 DELETE de proveidors, DELETE de subministraments
   - 1.3 DELETE "100, joan, ..." de proveidors
   - ....

Múltiples solucions!!! No és possible

2. INSERT "200, pere, ..., 200, m2, ..." a la vista =>

   INSERT "200, pere, ..." a proveidors, INSERT "200, m2, ..." a subministraments

Una solució! Hauria de ser possible

Esquema d'Informació

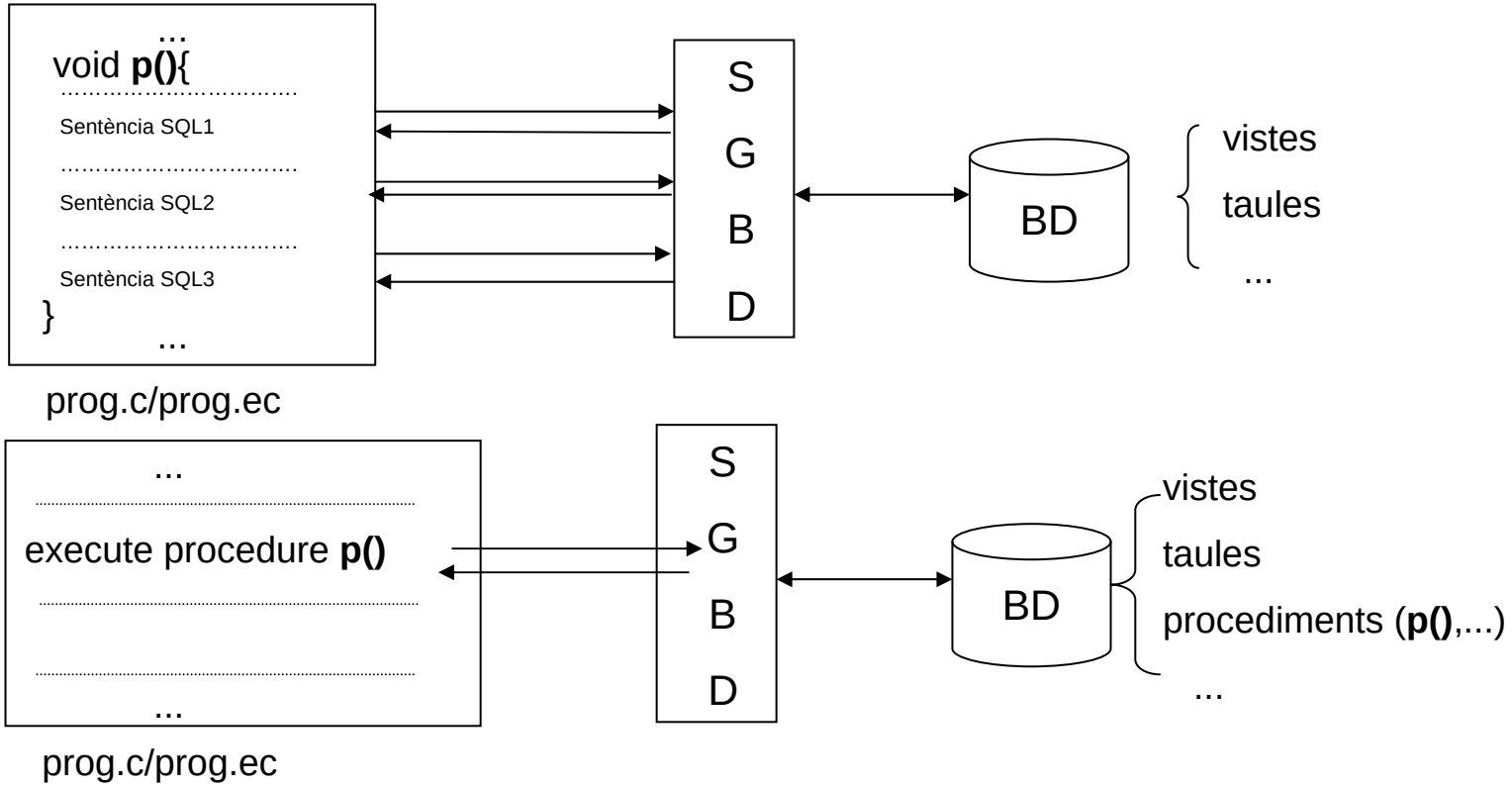
- Cada catàleg conté un esquema d'informació (*Information Schema*), a més dels esquemes definits pel propi usuari
- Tota la informació dels esquemes definits pels usuaris: noms i atributs de les taules, índexs, restriccions de columna, taula,... s'emmagatzema a la seva vegada a l'esquema d'informació. Així l'esquema d'informació és un esquema sobre esquemes: meta-dades !!
- L'esquema d'informació consisteix en un conjunt de vistes accesibles pels usuaris:
  - SCHEMATA: Informació de cada esquema del catàleg
  - DOMAINS: Informació sobre els dominis
  - TABLES: Informació sobre les taules
  - VIEWS: Informació sobre vistes
  - ASSERTIONS: Informació sobre restriccions
  - TRIGGERS: Informació sobre disparadors
  - ...
- Les vistes de l'esquema d'informació estan definides sobre un conjunt de *taules de sistema (definition schema)* accesibles només per l'administrador
- Altes, baixes i modificacions en aquestes taules de sistema són indirectes!!!

Procediments Emmagatzemats

- Un procediment emmagatzemat és una funció definida per un usuari que, un cop creat, s'emmagatzema a la BD i es tractat com un objecte més de la BD.
- Per poder escriure procediments emmagatzemats disposem de dos tipus de sentències:
  - Sentències pròpies de l'SQL
  - Sentències pròpies del llenguatge de programació que cada SGBD té per implementar els procediments emmagatzemats (e.x: PgSQL de PostgreSQL)
- Els procediments emmagatzemats es poden executar:
  - De manera interactiva, amb SQL directe o interactiu
  - Des d'una aplicació
  - Des d'un altre procediment emmagatzemat

Introducció

- Els procediments emmagatzemats serveixen per:
  - Simplificar el desenvolupament d'aplicacions
  - Millorar el rendiment de la BD
  - Controlar les operacions que els diferents usuaris realitzen contra la BD



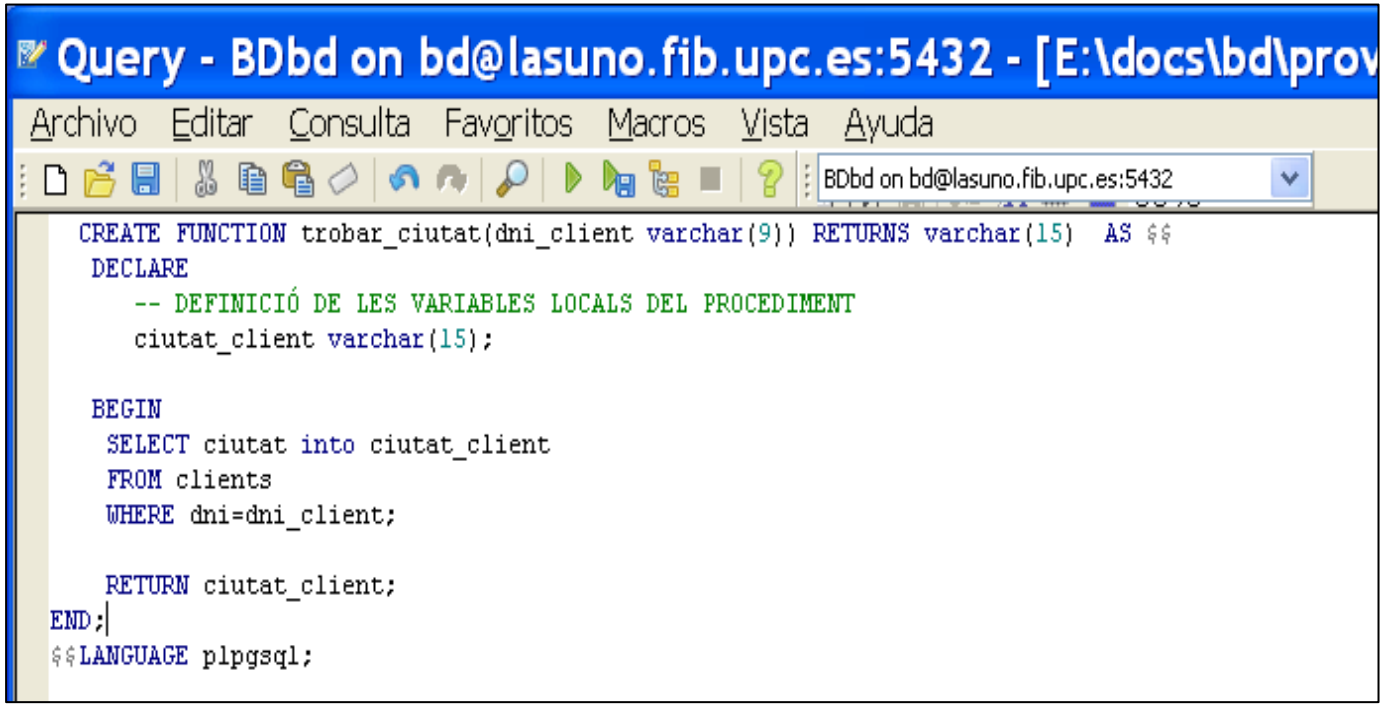
Introducció

Exemple de creació d'un procediment emmagatzemat:

```
CREATE FUNCTION trobar_ciutat(dni_client varchar(9))
RETURNS varchar(15)  AS $$
  DECLARE
     ciutat_client varchar(15);
  BEGIN
    SELECT ciutat INTO ciutat_client
    FROM clients
    WHERE dni=dni_client;

   RETURN ciutat_client;
END;
$$LANGUAGE plpgsql;
```

Introducció

Exemple de creació i d'execució d'un procediment emmagatzemat (1)



Introducció

Exemple de creació i d'execució d'un procediment emmagatzemat (2)

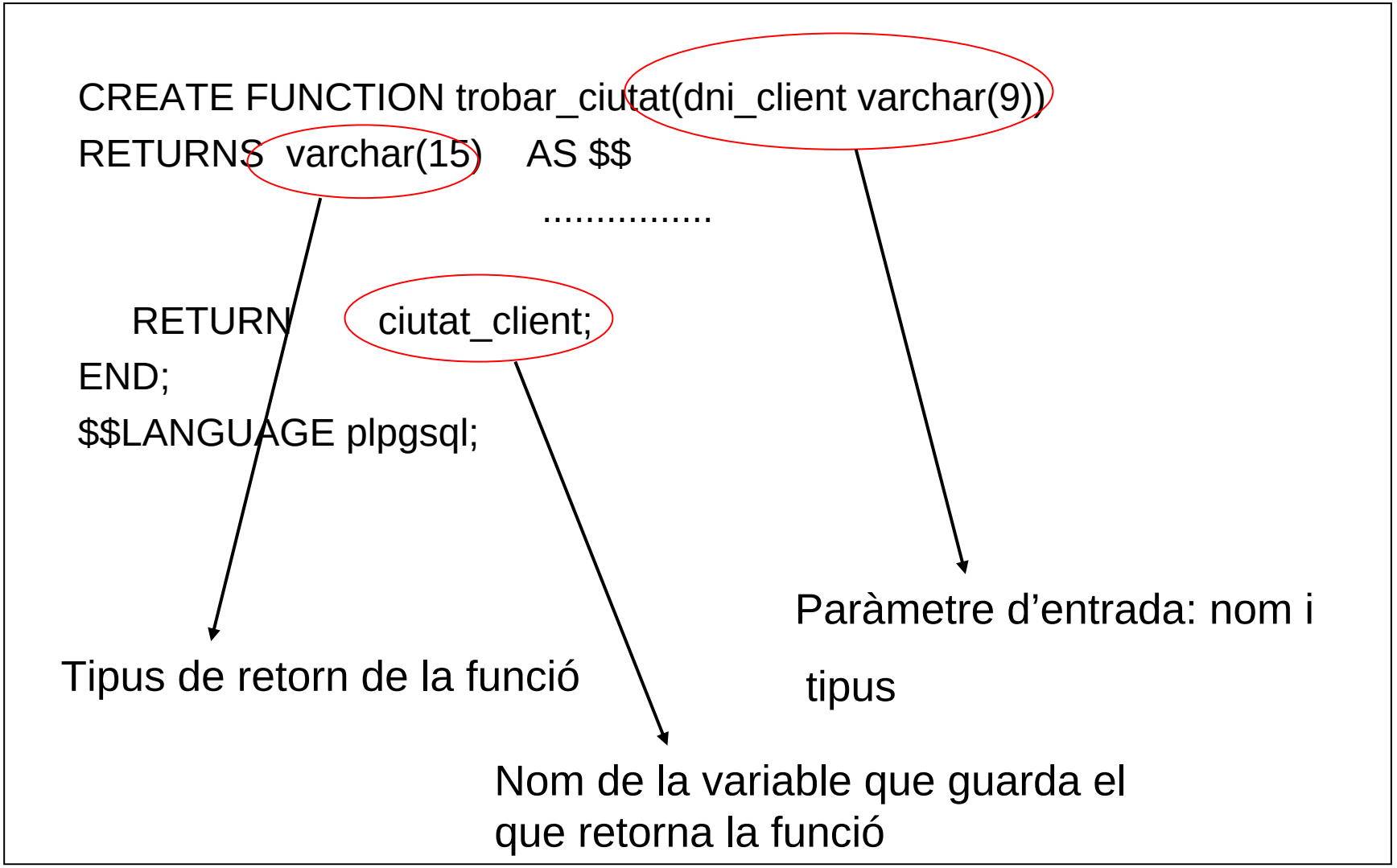
Per esborrar un procediment emmagatzemat:

```
DROP FUNCTION trobar_ciutat(varchar(9));
```

Introducció a PL/PGSQL

- PL/PGSQL és un dels llenguatges que ofereix postgreSQL per fer procediments emmagatzemats.
- Les sentències de PL/PGSQL les utilitzarem dins del cos d'un procediment, és a dir, entre les sentències CREATE FUNCTION i END de la funció.
- Bàsicament, PL/PGSQL proporciona dos tipus de sentències:
  - Sentències per definir (DECLARE) i assignar valors a variables
  - Sentències per controlar el flux d'execució d'un procediment:
    - Sentències condicionals:
      - Sentència IF
    - Sentències iteratives:
      - Sentències LOOP, FOR i WHILE
    - Sentències per fer la gestió d'errors:
      - Sentències EXCEPTION i RAISE EXCEPTION

Paràmetres


CREATE FUNCTION trobar_ciutat(dni_client varchar(9))
RETURNS varchar(15) AS $$
.....
RETURN ciutat_client;
END;
$$LANGUAGE plpgsql;
Tipus de retorn de la funció
Paràmetre d'entrada: nom i tipus
Nom de la variable que guarda el que retorna la funció

Variables

- Bàsicament, és possible utilitzar variables dins d'un procediment emmagatzemat en les situacions següents:
  - En sentències de l'SQL
  - En sentències de PL/PGSQL, per assignar, calcular valors i controlar el flux d'execució d'un procediment
- Definició i assignació de valors a variables:
  - La sentència DECLARE de PL/PGSQL ens permet definir variables dins d'un procediment
  - Si no s'inicialitzen les variables, per defecte prenen valor NULL
  - El valor d'una variable s'emmagatzema en memòria volàtil i per tant, no són considerades objectes de la BD
  - Totes les variables definides dins d'un procediment són variables locals.

    → L'àmbit de visibilitat d'una variable local queda restringit al procediment a on s'hagi definit

Variables

- Tipus de dades d'una variables:
  - Podem utilitzar els mateixos tipus de dades que els utilitzats a les columnes d'una taula.
  - També és possible especificar que el tipus de dades d'una variable és idèntic al tipus de dades d'una determinada columna d'una taula mitjançant la clàusula TYPE.

  La síntaxi general per definir una variable és:

  Nom_variable [CONSTANT] type [NOT NULL] [{ DEFAULT | := }expression];

- Exemples de definició de variables:

  nom_client char(15);

  carrer varchar(20) not null;

  edat integer default 18;

  num constant integer default 0;

  dni_client clients.dni%TYPE;

Variables

- Assignació de valors a variables: tenim tres possibilitats:
  - Sentència d'assignació de PL/PGSQL
  - Sentència SELECT ... INTO de l'SQL
  - Assignar a una variable el que retorna una funció

Exemples d'assignació de valors a variables:

- Sentència d'assignació de PL/PGSQL:

  numComclient=(SELECT num_com FROM clients WHERE dni=dni_client);
- Sentència SELECT ... INTO:

  SELECT ciutat INTO ciutat_client
  FROM clients
  WHERE dni=dni_client;
- Assignar a una variable el que retorna una funció:

  import_comanda:=import_una_com(cursor_com.num_com);

  o bé

  select * from import_una_com(cursor_com.num_com) into import_comanda;

Sentències condicionals

- La sentència IF serveix per a establir condicions en el flux d'execució d'un procediment:

  Sintaxi

  IF condició **THEN** bloc de sentències

  ELSE bloc de sentències

  END IF;

- Podem establir diferents nivells d'aniuament mitjançant la clàusula

  IF ... THEN ... ELSEIF ... THEN ... ELSE...END IF;

- Condicions:
  - Per especificar les condicions podem utilitzar:
    - Operadors lògics: AND, OR, NOT
    - Operadors de comparació: =, <,<=, >, >=, etc
    - Predicats propis d'SQL: BETWEEN, IN, IS NULL, LIKE
    - Consultes SQL

Sentències condicionals

Exemple d'utilització de sentències condicionals

```
CREATE FUNCTION calcul_desc_client(dni_client clients.dni%type) RETURNS integer AS $$
DECLARE
    descompte INTEGER;
    numComClient INTEGER;

BEGIN
   IF ((SELECT COUNT(*) FROM clients WHERE dni=dni_client)=1)THEN
      numComclient=(SELECT num_com FROM clients WHERE dni=dni_client);

      IF (numComClient IS NULL) THEN
          descompte=NULL;
      ELSIF (numComClient<5)  THEN
          descompte=0;
      ELSIF (numComClient<10) THEN
           descompte=3;
      ELSIF (numComClient<15) THEN
           descompte=5;
      ELSE
           descompte=10;
      END IF;
   END IF;
   RETURN descompte;
END;
$$LANGUAGE plpgsql;
```

Execució de la funció:

```
select * from calcul_desc_client('35678111')
as descompte;
```

Panel de Salida

Salida de datos | Comentar | Mensajes | Historial

	descompte integer
1	3

Retorn d'un conjunt d'atributs (1 única tupla)

Exemple: Obtenir l'adreça(carrer, num_carrer i ciutat) d'un client.

```
CREATE TYPE TAdressa AS (
   carrer varchar(20),
   num_carrer varchar(4),
   ciutat varchar(15)
);

drop FUNCTION trobar_adressa_client(varchar(9));
CREATE FUNCTION trobar_adressa_client (dni_client clients.dni%type) RETURNS TAdressa AS $$
DECLARE
      dadesCli TAdressa;
BEGIN
   SELECT carrer,num_carrer,ciutat INTO dadesCli
   FROM clients
   WHERE dni=dni_client;

   RETURN dadesCli;
END;
$$LANGUAGE plpgsql;

select * FROM trobar_adressa_client('45678900');
```

- Per retornar un conjunt d'atributs del client definim un nou tipus (Tadressa).
- Al procediment emmagatzemat cal definir una variable del tipus definit prèviament i assignar-li els valors corresponents.

Sentències iteratives

- PGSQL proporciona tres tipus de sentències iteratives:
  - Sentència **LOOP** :

    LOOP statements **EXIT** [**WHEN** expression]; **END LOOP**;
  - Sentència **WHILE**:

    WHILE expression **LOOP** statements **END LOOP**;

    La sentència WHILE i LOOP es poden utilitzar per definir bucles on el seu acabament estigui definit per una expressió condicional.
  - Sentència **FOR**
    - Es pot utilitzar quan sabem a priori el nombre d'iteracions a executar.

    FOR name **IN** [**REVERSE**] expression .. expression **LOOP**

    statements

    END LOOP;
    - També es pot utilitzar per iterar sobre el conjunt de tuples retornades per una consulta SQL.

      FOR target **IN** query **LOOP** statements **END LOOP**;

Sentències iteratives

BD d'exemple:

Clients(dni,nom,cognom1,cognom2,carrer,num_carrer,cp,ciutat)

Comandes(num_com,dni,data_arribada,import_total)

{dni} Referencia Clients

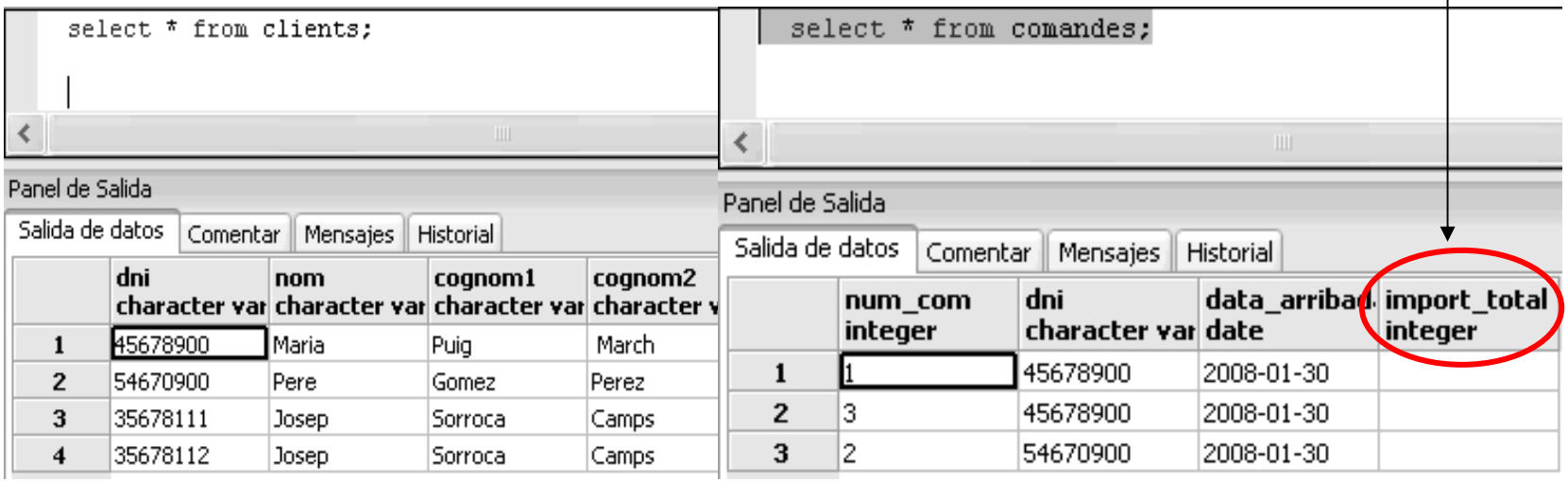
Items(num_item,preu_unitat)

Items_Comanda(num_item,num_com,qtt)

{num_item} Referencia Items

{num_com} Referencia Comandes

Atribut calculat



Sentències iteratives

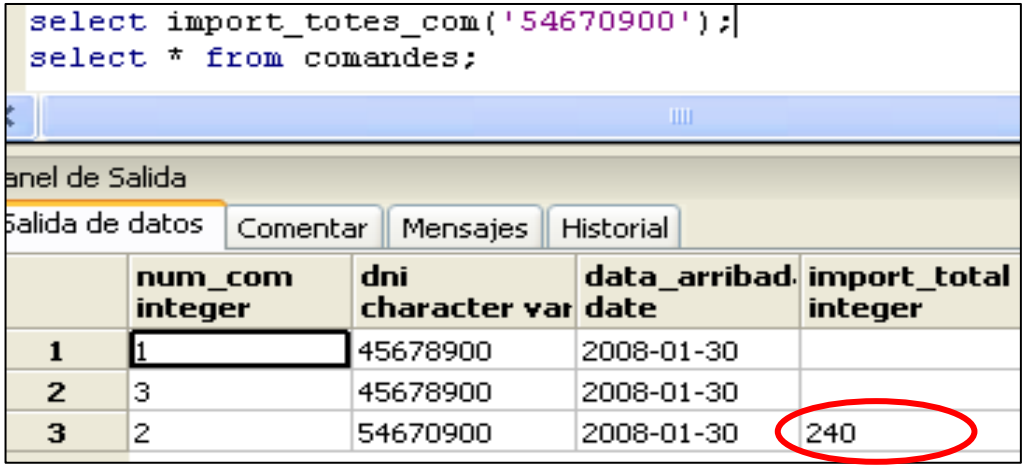
Exemple: donat un dni de client, calcular l'import total de les comandes d'aquest client. Utilitzarem un bucle FOR per accedir al resultat d'una consulta.

```
drop function   import_una_com(com_client comandes.num_com%TYPE);
CREATE FUNCTION import_una_com(com_client comandes.num_com%TYPE) RETURNS integer AS $$
DECLARE
  total_com integer;
  dades_item_preu items.preu_unitat%type;
  dades_item_qtt items_comanda.quantitat%type;
BEGIN
 total_com=0;
 FOR dades_item_qtt,dades_item_preu IN SELECT ic.quantitat,i.preu_unitat
                                       FROM items_comanda ic, items i
                                       WHERE i.num_item=ic.num_item AND ic.num_com=com_client
                                       LOOP
        total_com=total_com+(dades_item_qtt*dades_item_preu);
 END LOOP;
 RETURN total_com;
END;
$$LANGUAGE plpgsql;

drop function import_totes_com(dni_client clients.dni%type);
CREATE FUNCTION import_totes_com(dni_client clients.dni%type) RETURNS void AS $$
DECLARE
  num_comanda comandes.num_com%type;
  import_comanda comandes.import_total%type;
BEGIN
FOR num_comanda IN SELECT num_com FROM comandes
                   WHERE dni=dni_client LOOP
        import_comanda:=import_una_com(num_comanda);
        UPDATE comandes set import_total=import_comanda
        WHERE num_com=num_comanda;

END LOOP;
END;
$$LANGUAGE plpgsql;
```

Sentències iteratives

Exemple: donat un dni de client calcular l'import total de les comandes d'aquest client. Utilitzarem un bucle FOR per accedir al resultat d'una consulta.

Execució de la funció:



	num_com integer	dni character var	data_arribad date	import_total integer
1	1	45678900	2008-01-30	
2	3	45678900	2008-01-30	
3	2	54670900	2008-01-30	240

Sentències iteratives

Retorn d'un conjunt de files (1): Tipus de dades simple

```
drop FUNCTION trobar_ciutat_client(nom_client clients.nom%type);
CREATE FUNCTION trobar_ciutat_client (nom_client clients.nom%type) RETURNS SETOF clients.ciutat%type AS $$
DECLARE
  ciutat_clients clients.ciutat%type;
BEGIN
 FOR ciutat_clients IN  SELECT ciutat
                        FROM clients
                        WHERE nom=nom_client LOOP

  RETURN NEXT ciutat_clients;
 END LOOP;

END;
$$LANGUAGE plpgsql;

select * FROM trobar_ciutat_client('Josep');
```

anel de Salida

Salida de datos | Comentar | Mensajes | Historial

	trobar_ciutat_client character varying
1	Madrid
2	Girona

•Per retornar un conjunt de files cal utilitzar SETOF quan especifiquem el tipus que retorna la funció. Combinem la sentència FOR ... Amb RETURN NEXT per aconseguir que un procediment retorni un conjunt de files o resultats.

Sentències iteratives

```
CREATE TYPE Tdades_client AS (
        dni_client VARCHAR(9),
        nom_client VARCHAR(15),
        cognom1 VARCHAR(15));

drop function clients_ciutat(ciutat_client clients.ciutat%type);
CREATE FUNCTION clients_ciutat(ciutat_client clients.ciutat%type) RETURNS SETOF Tdades_client AS $$
DECLARE
 cursor_clients Tdades_client;
BEGIN
FOR cursor_clients IN SELECT dni,nom,cognom1
                      FROM clients
                      WHERE ciutat=ciutat_client LOOP
 return next cursor_clients;
END LOOP;
END;
$$LANGUAGE plpgsql;

select * FROM clients_ciutat('Barcelona');
```

Panel de Salida

Salida de datos | Comentar | Mensajes | Historial

	dni_client character var	nom_client character var	cognom1 character var
1	45678900	Maria	Puig
2	54670900	Pere	Gomez

Sentències iteratives

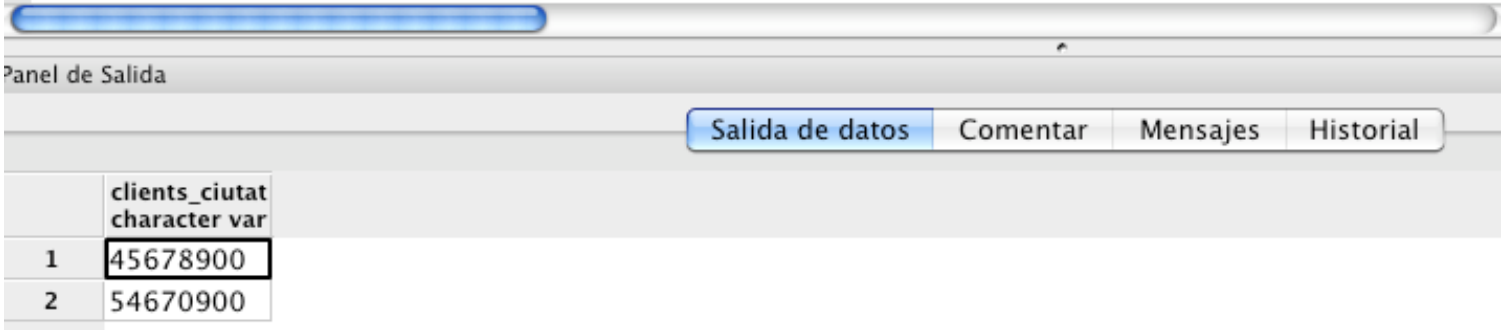
Cursors explícits (1).

- Quan s'executa una sentència FOR per accedir al conjunt de resultats d'una consulta SQL, PostgreSQL utilitza implícitament cursors.
  - Un cursor és una estructura de dades que permet accedir a cada una de les files que retorna una consulta SQL.
  - Addicionalment PostgreSQL permet treballar amb cursors explícits per a accedir a cada una de les files retornades per una consulta SQL (nosaltres no ho farem servir). Per treballar amb cursors explícits cal:
    1. Declarar el cursor i associar-lo a una consulta (DECLARE ... CURSOR FOR)
    2. Obrir el cursor (OPEN)
    3. Obtenir la primera fila de la consulta associada al cursor (FETCH)
    4. Executar el bloc de sentències del bucle
    5. Obtenir la següent fila de la consulta associada al cursor (FETCH)
    6. Tornar a executar els passos 4) i 5) mentre hi hagin files que formin part del resultat de la consulta.
    7. Tancar el cursor (CLOSE)

Sentències Iteratives

Cursors explícits (2). Exemple d'utilització.

```
CREATE FUNCTION clients_ciutat(ciutat_client clients.ciutat%type) RETURNS SETOF varchar(9) AS $$
DECLARE
  cursor_clients CURSOR FOR SELECT dni
                            FROM clients
                            WHERE ciutat=ciutat_client;
  dni_client clients.dni%type;
BEGIN
OPEN cursor_clients;
LOOP
   FETCH cursor_clients INTO dni_client;
   EXIT WHEN NOT FOUND;
   return next dni_client;
END LOOP;
close cursor_clients;
END;
$$LANGUAGE plpgsql;

select * FROM clients_ciutat('Barcelona');
```



Sentències iteratives

Exemple d'utilització de la sentència WHILE

```
drop function incrementar_preu();
CREATE FUNCTION incrementar_preu()RETURNS void as $$
BEGIN
WHILE EXISTS(SELECT * FROM items WHERE preu_unitat<25) LOOP
  UPDATE items
  SET preu_unitat=preu_unitat+5
  WHERE preu_unitat<25;
END LOOP;
END;
$$LANGUAGE plpgsql;
```

Contingut de la taula Items abans i després d'executar el procediment.

Que hauria passat si en comptes d'utilitzar un WHILE haguéssim utilitzat un FOR ?

	num_item integer	preu_unitat integer
1	1	15
2	2	20
3	3	15
4	4	20
5	5	15
6	6	50
7	7	40

	num_item integer	preu_unitat integer
1	1	25
2	2	25
3	3	25
4	4	25
5	5	25
6	6	50
7	7	40

Gestió d'errors

Quan es produeix un error dintre d'un procediment podem:

• **No capturar-lo**: El procediment falla i es retorna l'error concret al nivell superior (JDBC, editor d'SQL, a un altre procediment..)

•**Capturar-lo:**

- El procediment falla i es retorna una **excepció** determinada pel programador al nivell superior.
- El procediment té èxit i es retorna un **codi d'error** determinat pel programador al nivell superior.
- El procediment té èxit i s'insereix l'error en un **taula d'errors.**
- El procediment té èxit i l'error es tractat dins del procediment.

Gestió d'errors

Tipus d'errors que es poden produir:

- Errors predefinits pel propi SGBD, p.e. el codi d'error número 23505 a PostgreSQL vol dir "Unique violation"
- Errors específics del procediment (P0001 a PostgreSQL). Aquests errors seran generats pel creador del procediment emmagatzemat.

PostgreSQL proporciona dos instruccions per gestionar els errors:

- **EXCEPTION**: accions a dur a terme en cas de que es produeixin errors.

- **RAISE EXCEPTION**: serveix per a que el programador pugui generar els seus propis errors dins d'un procediment

Gestió d'errors

- Quan un procediment falla, PostgreSQL aborta l'execució de la funció i de la transacció en curs.
- Es pot utilitzar el bloc BEGIN… amb la claúsula EXCEPTION per tractar els errors que es produeixen dins del procediment.

  BEGIN

  statements

  EXCEPTION

  WHEN *condition* [OR *condition* ...] THEN

  handler_statements

  [WHEN *condition* [OR *condition* ...] THEN

  handler_statements ...]

  END;
- Si durant l'execució del bloc de sentències associat a la sentència EXCEPTION es produeix un error, l'execució del procediment finalitza i l'error es reportat a l'usuari o programa que ha provocat l'execució del procediment emmagatzemat.

Gestió d'errors: Excepcions

```
drop function calcul_desc_client(dni_client clients.dni%type);
CREATE FUNCTION calcul_desc_client(dni_client clients.dni%type) RETURNS integer AS $$
DECLARE
    descompte INTEGER;
    numComClient INTEGER;
BEGIN
   IF ((SELECT COUNT(*) FROM clients WHERE dni=dni_client)=1)THEN
      numComclient=(SELECT num_com FROM clients WHERE dni=dni_client);

      IF (numComClient IS NULL) THEN
          RAISE EXCEPTION 'El client % no té comandes',dni_client;
      ELSIF (numComClient<5)  THEN
          descompte=0;
      ELSIF (numComClient<10) THEN
           descompte=3;
      ELSIF (numComClient<15) THEN
           descompte=5;
      ELSE
           descompte=10;
      END IF;
   ELSE
      RAISE EXCEPTION 'El client % no existeix',dni_client;
   END IF;

   RETURN descompte;

   EXCEPTION
      WHEN raise_exception THEN
           RAISE EXCEPTION' %: %',SQLSTATE, SQLERRM;
      WHEN OTHERS THEN
           RAISE EXCEPTION' P0001: Error intern';
END;
$$LANGUAGE plpgsql;
```

Execució de la funció generant una excepció

```
select calcul_desc_client('45678900') as descompte;
```

Panel de Salida

Salida de datos | Comentar | Mensajes | Historial

```
ERROR:   P0001: El client 45678900 no té comandes

********** Error **********

ERROR:  P0001: El client 45678900 no té comandes
Estado SQL:P0001
```

```
select * from clients;
```

anel de Salida

alida de datos | Comentar | Mensajes | Historial

	dni character var	nom character var	cognom1 character var	cognom2 character var	carrer character var	num_carrer character var	cp character(5)	ciutat character var	num_com integer
1	45678900	Maria	Puig	March	Valencia	123	08023	Barcelona	
2	54670900	Pere	Gomez	Perez	Enamorats	109	08032	Barcelona	2
3	35678111	Josep	Sorroca	Camps	Numancia	22	28023	Madrid	7
4	35678112	Josep	Sorroca	Camps	Numancia	22	28023	Madrid	7

Gestió d'errors: Codi de retorn

```
CREATE TYPE Tdesc_client AS(
   descompte integer,
   sql_state varchar(5),
   miss_error varchar(50)
);

drop function calcul_desc_client(dni_client clients.dni%type);
CREATE FUNCTION calcul_desc_client(dni_client clients.dni%type) RETURNS Tdesc_client AS $$
DECLARE
    descompte INTEGER;
    numComClient INTEGER;
    descompte_client Tdesc_client;
BEGIN
   IF ((SELECT COUNT(*) FROM clients WHERE dni=dni_client)=1)THEN
      numComclient=(SELECT num_com FROM clients WHERE dni=dni_client);

      IF (numComClient IS NULL) THEN
          RAISE EXCEPTION 'El client % no té comandes',dni_client;
      ELSIF (numComClient<5)  THEN
          descompte=0;
      ELSIF (numComClient<10) THEN
           descompte=3;
      ELSIF (numComClient<15) THEN
           descompte=5;
      ELSE
           descompte=10;
      END IF;
   ELSE
      RAISE EXCEPTION 'El client % no existeix',dni_client;
   END IF;

   descompte_client.sql_state=SQLSTATE;
   descompte_client.miss_error='';
   descompte_client.descompte=descompte;
   RETURN descompte_client;

   EXCEPTION
      WHEN raise_exception THEN
           descompte_client.sql_state=SQLSTATE;
           descompte_client.miss_error=SQLERRM;
           descompte_client.descompte=-1;
           RETURN descompte_client;

      WHEN OTHERS THEN
           descompte_client.sql_state=SQLSTATE;
           descompte_client.miss_error='Error intern';
           descompte_client.descompte=-1;
           RETURN descompte_client;

END;
$$LANGUAGE plpgsql;
```

Execució de la funció generant un error.

```
select * from calcul_desc_client('45678900') as descompte;
```

anel de Salida

Salida de datos | Comentar | Mensajes | Historial

	descompte integer	sql_state character var	miss_error character varying(50)
1	-1	P0001	El client 45678900 no té comandes

Gestió d'errors: Taula d'errors

```
CREATE TABLE prova(
  a integer primary key,
  b integer);

CREATE TABLE taula_errors(
   error_id integer primary key,
   error_code VARCHAR(5),
   proc_error VARCHAR(15),
   missatge_errror VARCHAR(100));


drop function insercions(d1 integer, d2 integer);
CREATE FUNCTION insercions(d1 integer, d2 integer) returns void  AS $$
DECLARE
  darrer_error integer;
BEGIN
  INSERT INTO prova VALUES (d1,d2);
EXCEPTION
   WHEN OTHERS THEN
     SELECT COUNT(*) INTO darrer_error FROM taula_errors;
     darrer_error=darrer_error+1;
     INSERT INTO taula_errors VALUES (darrer_error,SQLSTATE,'insercions',SQLERRM);
END;
$$LANGUAGE plpgsql;

select * from insercions(1,1);
select * from insercions(1,1);
select * from taula_errors;
```

Panel de Salida

Salida de datos | Comentar | Mensajes | Historial

	error_id integer	error_code character var	proc_error character var	missatge_errror character varying(100)
1	1	23505	insercions	duplicate key violates unique constraint "prova_pkey"

•En aquest exemple la clàusula EXCEPTION gestiona qualsevol error predefinit que es pugui produir durant l'execució del procediment.

•En cas de produir-se un error, deixem constància de la seva existència a taula_errors i l'execució del procediment finalitza. Si no es produeixen errors, el procediment finalitza la seva execució sense inserir res a la taula d'errors.

Gestió d'errors: L'error es tracta dins del procediment

```
drop function insercions(d1 integer, d2 integer);
CREATE FUNCTION insercions(d1 integer, d2 integer) returns void  AS $$

BEGIN
  INSERT INTO prova VALUES (d1,d2);
EXCEPTION
   WHEN UNDEFINED_TABLE THEN
      CREATE TABLE prova(
                  a integer primary key,
                  b integer);

END;
$$LANGUAGE plpgsql;
```

- El procediment intenta recuperar-se de l'error que s'ha produit.
- En aquest exemple, si la taula prova no existeix, es produeix un error. El procediment intenta solucionar-lo creant la taula.
- ALERTA: Això no sempre es pot fer !!

Execució de sentències de crida a procediments emmagatzemats

```
Create table A(x integer primary key,y varchar(10));
Create function C (y varchar(10)) Returns int AS $$
Declare
   aux int;
Begin
 Insert into A Values (1,y);
 Select Max(x) into aux From A;
 Return aux;
END;
$$LANGUAGE plpgsql;
```

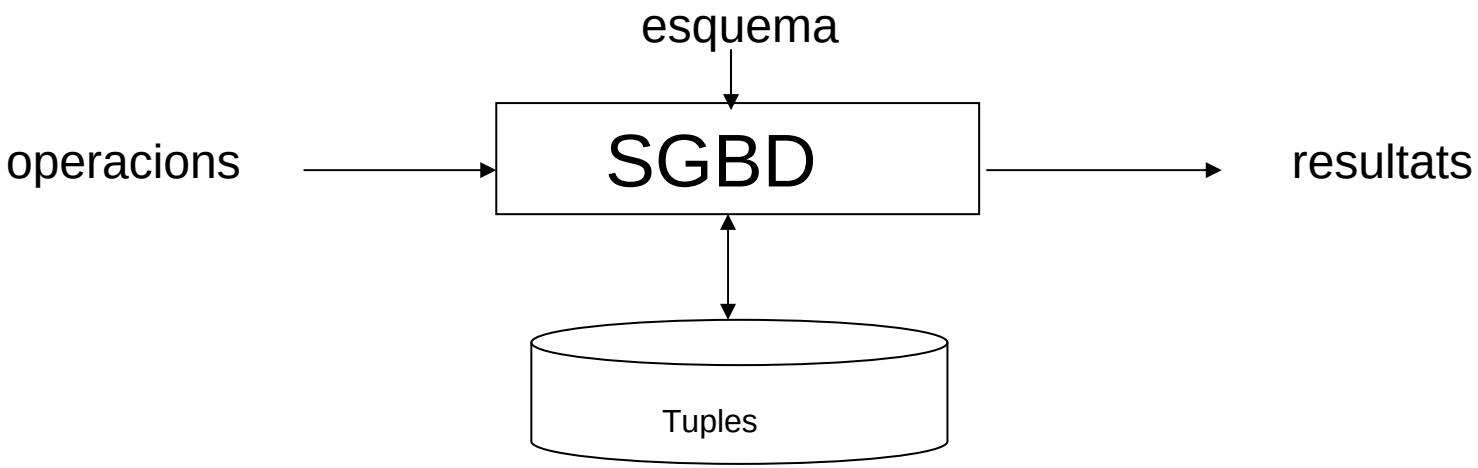
// Fragment de codi Java

```
Statement s=null;
CallableStatement cs=null;

try
{
   // creem un Statement
   s = c.createStatement ();

   // creem el CallableStatement
    cs = c.prepareCall ("{? = call C (?)}");

    cs.setString (1,"David");
    ResultSet rs = cs.executeQuery ();

    rs.next();

    int codi = rs.getInt (1);
    System.out.println ("David te el codi "+codi);

}
catch (SQLException se)
{
   System.out.println ("Error a l'executar les
   sentencies.");
}
```

DISPARADORS

- Motivació: Sistemes Passius vs Actius
- Disparadors en PostgreSQL
- Consideracions de disseny

SGBD Convencionals: principis generals



Representació de la semàntica del món real dintre d'un marc de:

- Model de dades
  - Estructura per dades de tipus tuples
  - Operacions per manipular-les
  - (algunes) regles d'integritat

- Model de transaccions
  - Execució "sota demanda" de transaccions definides pels usuaris amb garantia de compliment d'alguns criteris (ACID)

Altres tipus de semàntiques

- Monitorització/alerta de determinades situacions → comportament reactiu

  (Ex: auditoria d'operacions,

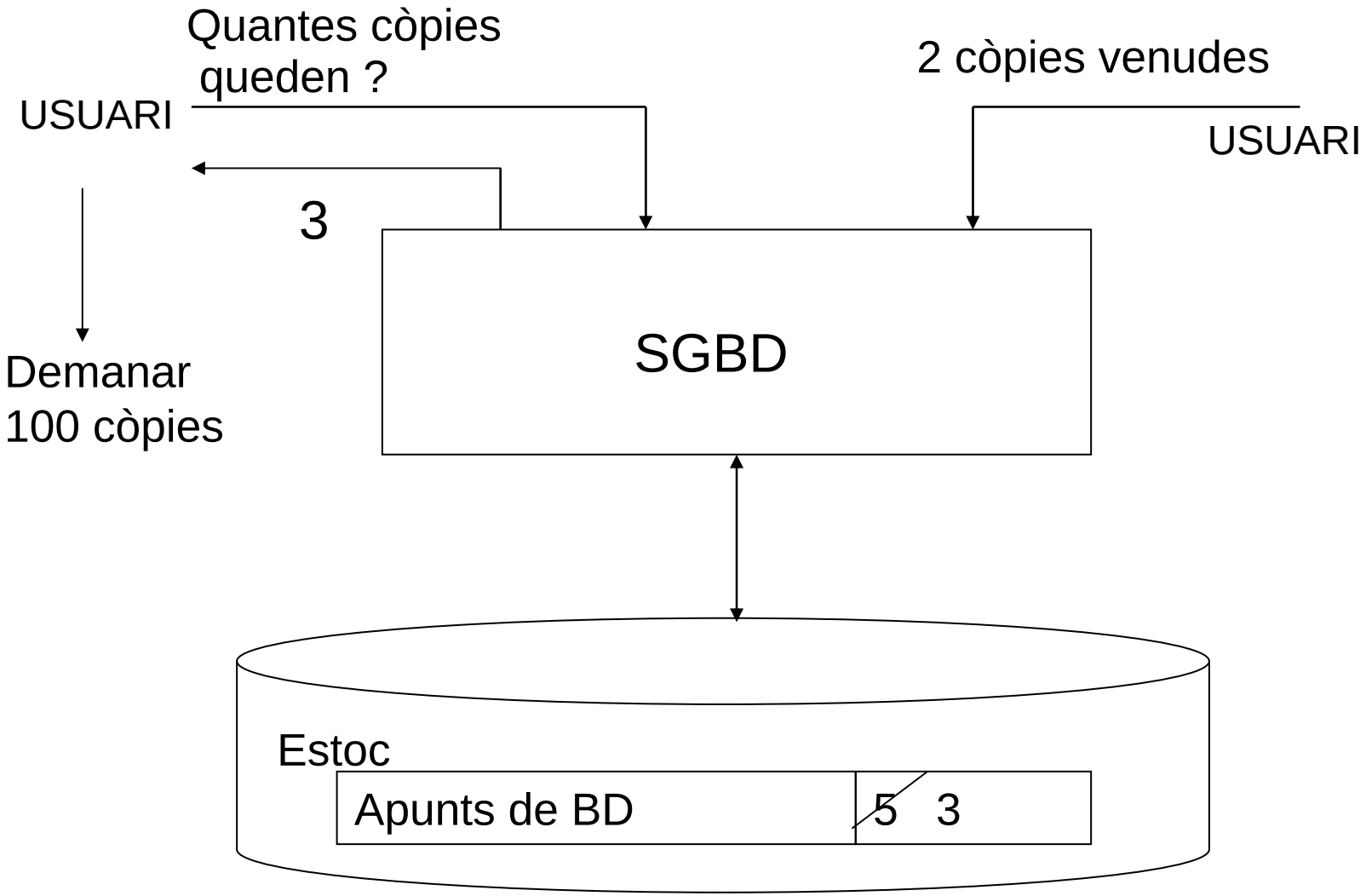
  avisos de situacions,

  regles de negoci)
- Comprovació de restriccions d'integritat no expressables directament en el model

  (Ex: restriccions dinàmiques, assercions, ...)
- Manteniment de restriccions d'integritat
- Manteniment automàtic:
  - d'atributs derivats
  - vistes materialitzades
  - ...

⇒ Poc suport dels sistemes convencionals !!!

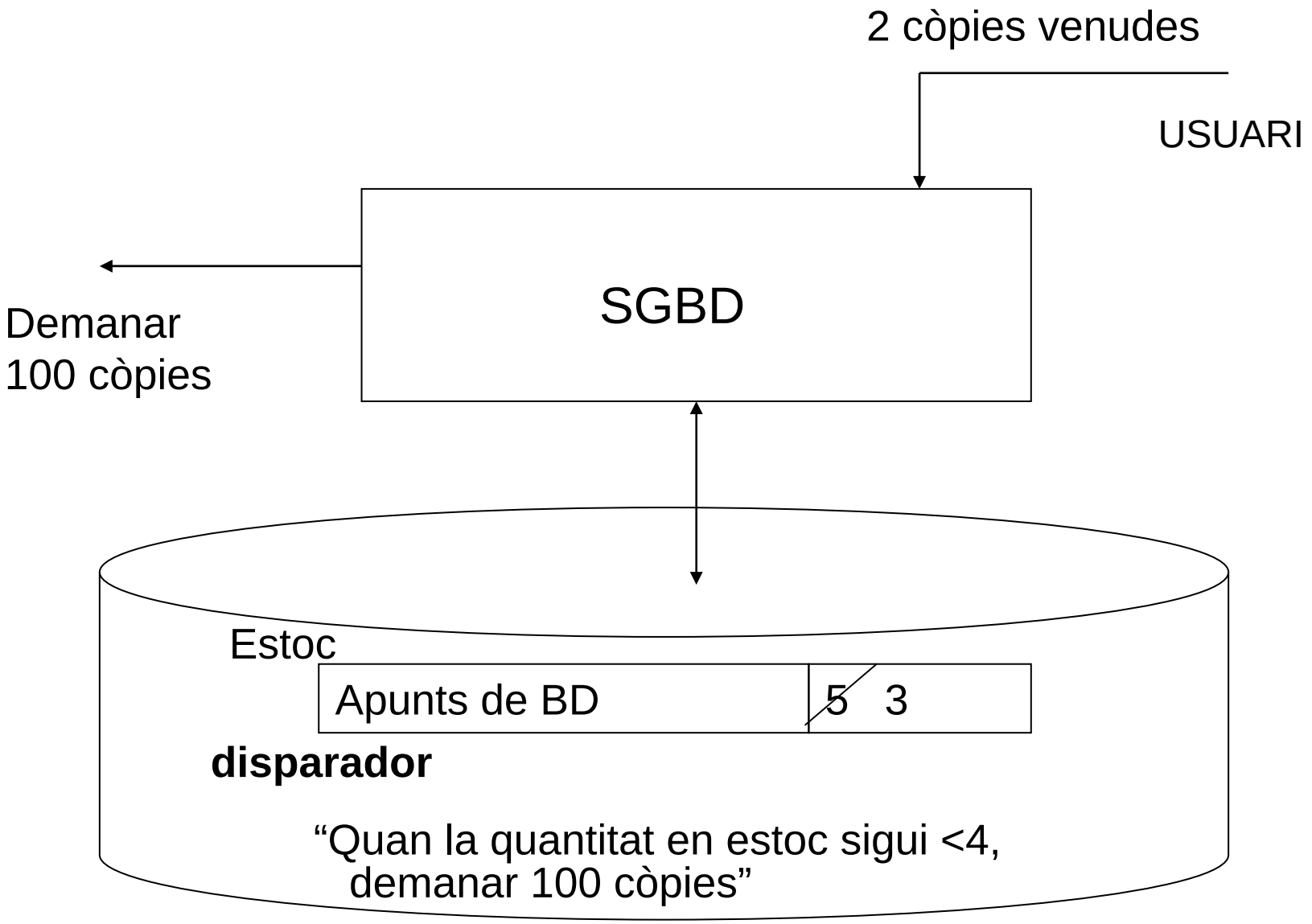
Els SGBD convencionals son “passius”



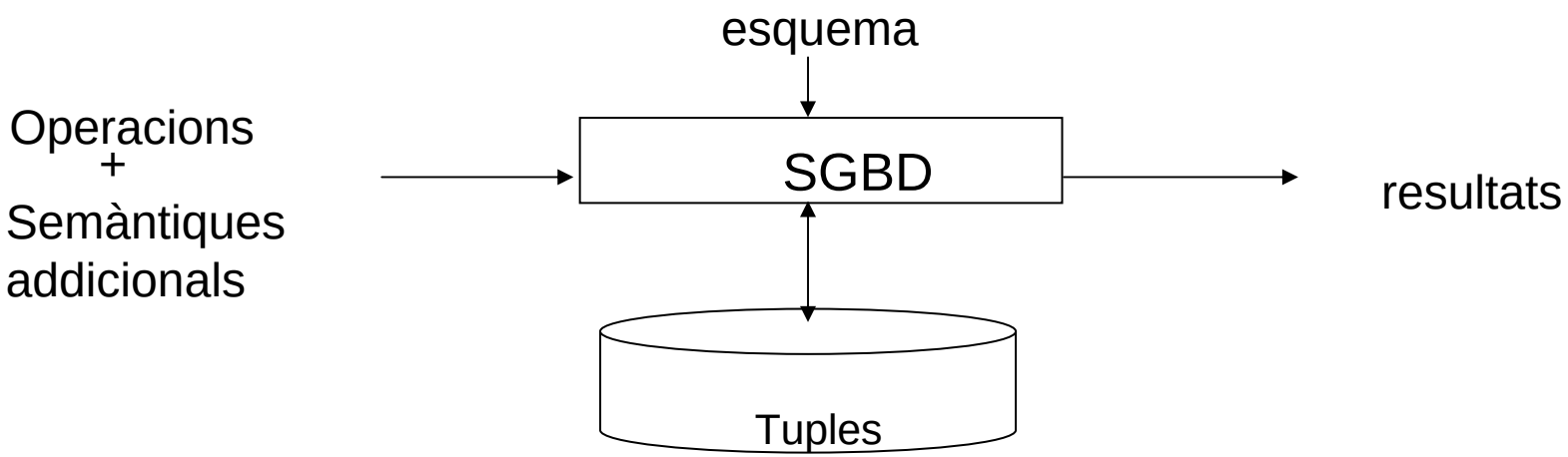
Solucions Convencionals

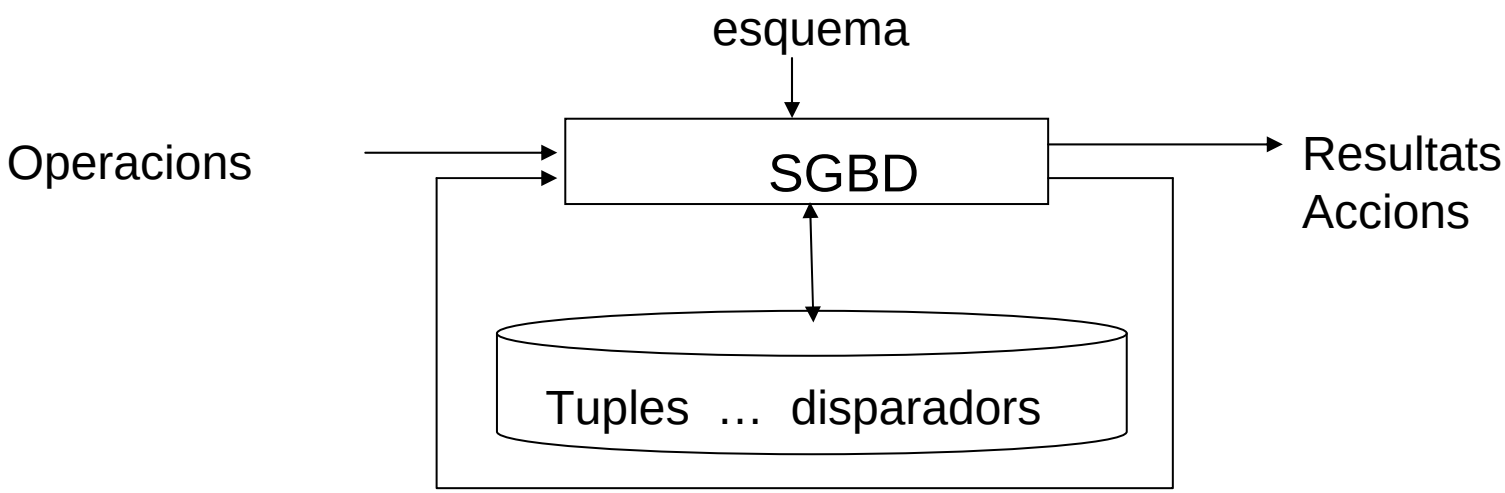
- Els programes/transaccions es preocupen de les semàntiques addicionals
  - La semàntica està:
    - Amagada en el codi dels programes
    - Distribuïda/replicada en molts programes
    - (Difícil de trobar, canviar, …)
  - Eficàcia/correctesa depèn de cada programa/programador
- “Polling” periòdic de la BD (usant programes dedicats)
  - La semàntica està concentrada en un lloc
  - “Polling” ocasionals: podem perdre el bon moment per reaccionar
  - “Polling” freqüent: pèrdua d’eficiència

Activitat


2 còpies venudes
USUARI
SGBD
Demanar
100 còpies
Estoc
Apunts de BD
5 3
disparador
“Quan la quantitat en estoc sigui <4,
demanar 100 còpies”

SGBD Passius vs Actius





DISPARADORS

Un disparador és una regla ECA:

Esdeveniment E

Condició C

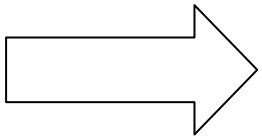
Acció A

“Quan es produeix E, si C, aleshores executar A”

La majoria dels sistemes comercials disposen de disparadors:

Oracle, PostgreSQL, SQL Server

I a SQL:1999 es defineixen com a estàndard.

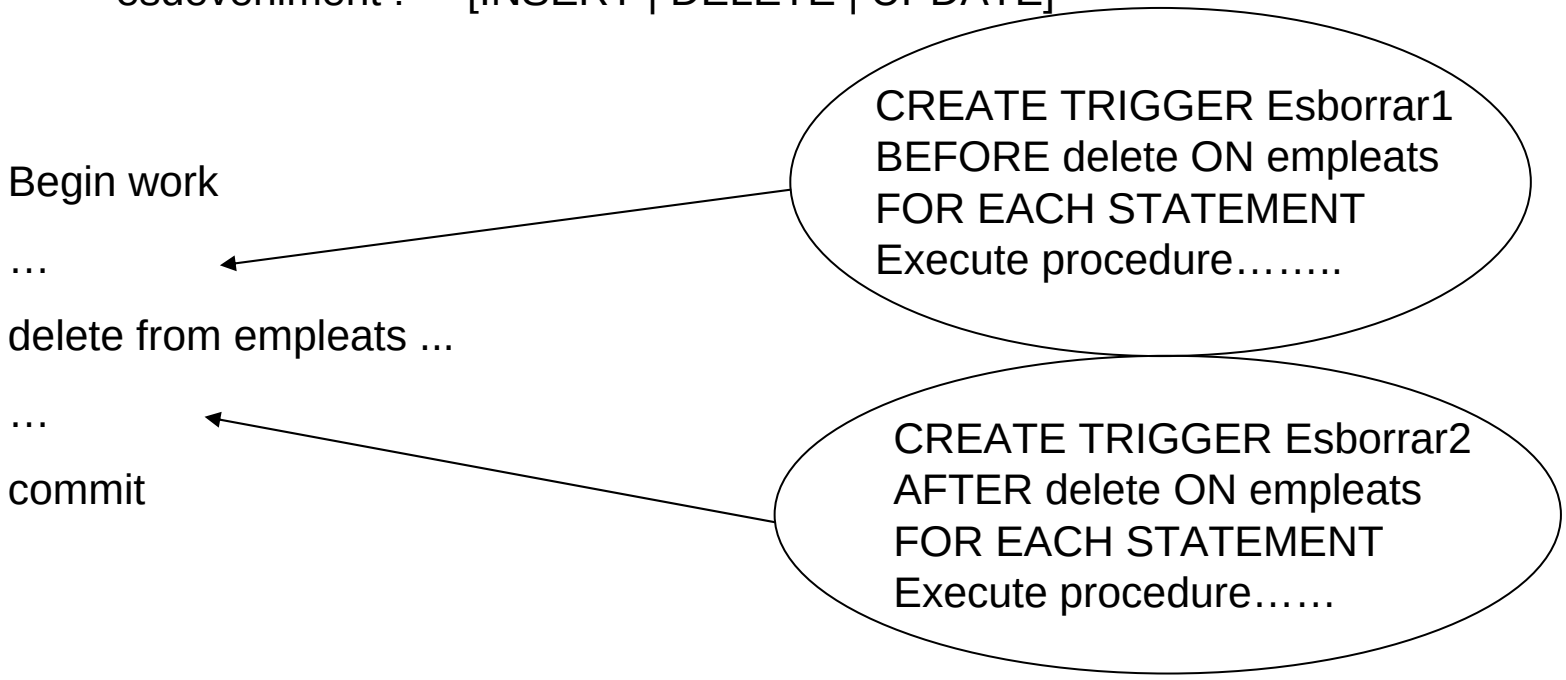


petites diferències entre els sistemes !!!

Disparadors en PostgreSQL: Sintaxi

```
CREATE TRIGGER nom { BEFORE | AFTER }
{ esdeveniment [ OR ... ] } ON taula
 [ FOR [ EACH ] { ROW | STATEMENT } ]
 EXECUTE PROCEDURE nomFunc ( arguments )
```

esdeveniment :== [INSERT | DELETE | UPDATE]



Disparadors a PostgreSQL: Ordre d'execució i visibilitat

1. BEFORE STATEMENT :

   L'acció s'executa **1 sola vegada abans de l'execució de la sentència** que dispara el disparador.

2. BEFORE ROW

   L'acció s'executa **1 vegada per a cada tupla** afectada i just abans que la tupla s'insereixi, modifiqui o esborri.

3. AFTER ROW

   L'acció s'executa **1 vegada per a cada tupla** afectada i després de l'execucuió de la sentència que dispara el disparador.

4. AFTER STATEMENT

   L'acció s'executa **1 sola vegada després de l'execució de la sentència** que dispara el disparador.

Exemple -1: Auditoria

- Objectiu: mantenir un registre de les modificacions que fan els usuaris a la taula Items. Cada vegada que es modifiqui la quantitat d'un item haurem de guardar un registre amb: ident. item, usuari que ha fet la modificacio, data, quantitat inicial i quantitat final de item a la taula log_record.

```
CREATE TABLE log_record(
  item integer,
  username char(8),
  update_time timestamp,
  old_qtt integer,
  new_qtt integer);

drop table items;
create table items(
  item integer primary key,
  name char(25),
  qtt integer,
  preu_total decimal(9,2));

insert into items values(1,'sac',100,300);
insert into items values(2,'bolis',5000,50000);
insert into items values(3,'rats',500,5000);


drop function insert_log();
CREATE FUNCTION insert_log() RETURNS trigger AS $$
    BEGIN
      if (OLD.qtt<>NEW.qtt) THEN
         insert into log_record values (OLD.item,current_user,current_date,OLD.qtt,NEW.qtt);
      END IF;
       RETURN NULL;
    END;
$$ LANGUAGE plpgsql;


CREATE TRIGGER restrict_after_empl
AFTER UPDATE ON items
FOR EACH ROW EXECUTE PROCEDURE insert_log();
```

Exemple -1: Execució

Contingut inicial de la taula items

```
select * from items;
```

anel de Salida

Salida de datos | Comentar | Mensajes | Historial

	item integer	name character(25	qtt integer
1	1	sac	100
2	2	bolis	5000
3	3	rats	500

Contingut de la taula log_record després d'executar-se les accions del disparador.

```
update items set qtt=qtt+10 where item<>3;
select * from log_record;
```

anel de Salida

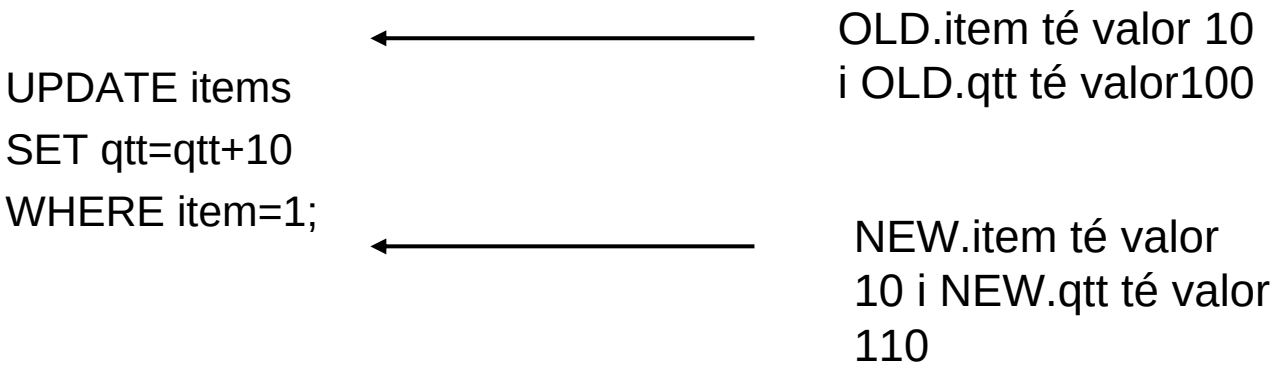
Salida de datos | Comentar | Mensajes | Historial

	item integer	username character(8)	update_time timestamp wi	old_qtt integer	new_qtt integer
1	1	bd	2008-02-04 00:(	100	110
2	2	bd	2008-02-04 00:(	5000	5010

Variables

- **OLD**: tupla abans de l'execució d'una sentència update o delete per triggers for each row. NULL en cas de for each statement.

- **NEW**: tupla després de l'execució d'una sentència update o insert per triggers for each row. NULL en cas de for each statement.



- **TG_OP**: conté l'esdeveniment que llança el disparador. Els seus valors poden ser (en majúscules!): INSERT, DELETE i UPDATE.
- Hi ha altres variables!. Consultar el manual de postgres.

Exemple -2: Auditoria

Objectiu: mantenir un registre de les modificacions que fan els usuaris a la taula Items. Ara, només voldrem guardar l'usuari que fa la modificació i la data en que es produeix la modificació.

```
DROP TABLE log_record2;
CREATE TABLE log_record2(
  username char(8),
  update_time timestamp);
commit;

drop function inserta_log();
CREATE FUNCTION inserta_log() RETURNS trigger AS $$
    BEGIN
         insert into log_record2 values (current_user,current_date);
       RETURN NULL;
    END;
$$ LANGUAGE plpgsql;


CREATE TRIGGER audit_items
AFTER UPDATE ON items
FOR EACH STATEMENT EXECUTE PROCEDURE inserta_log();

update items set qtt=qtt+10 where item<>3;
select * from log_record2;
```

Panel de Salida

Salida de datos | Comentar | Mensajes | Historial

	username character(8)	update_time timestamp witho
1	bd	2008-02-01 00:00:0(

Exemple 3: Atribut derivat

- Objectiu: Donada la taula items mantenir de manera automàtica l'atribut derivat preu_total quan hi ha modificacions de la quantitat d'estoc.

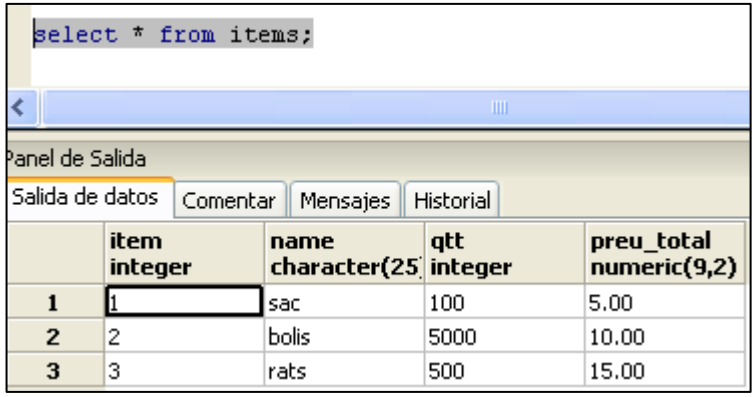
```
drop function calcular_nou_total();
CREATE FUNCTION calcular_nou_total()RETURNS trigger AS $$
BEGIN
  IF (OLD.qtt<>NEW.qtt) THEN
   IF (old.qtt<>0) THEN
      NEW.preu_total=((OLD.preu_total/OLD.qtt)*NEW.qtt);
   END IF;
  END IF;
  RETURN NEW;
END
$$ LANGUAGE plpgsql;

drop trigger atribut_derivat on items;
CREATE TRIGGER atribut_derivat BEFORE UPDATE ON items
    FOR EACH ROW EXECUTE PROCEDURE calcular_nou_total();
```

•Les funcions invocades per disparadors for each row / before han de retornar la variable New (en cas d'un disparador per insert o update) o OLD (en cas de delete). Els altres tipus de disparadors han de retornar NULL.

• Motivació: Els disparadors for each row / before poden modificar les dades que s'inseriran / modificaran / esborraran modificant aquestes variables.

Exemple 3: Execució

Contingut inicial de la taula items

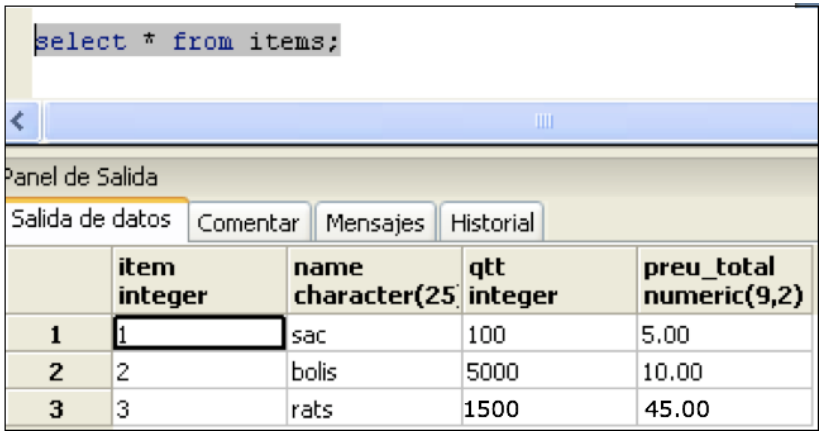


	item integer	name character(25	qtt integer	preu_total numeric(9,2)
1	1	sac	100	5.00
2	2	bolis	5000	10.00
3	3	rats	500	15.00

Sentència que dispara l'execució del disparador:

```
UPDATE items
SET qtt=qtt+1000
WHERE item=3;
```

Contingut final de la taula items després d'executar-se les accions del disparador -->



	item integer	name character(25	qtt integer	preu_total numeric(9,2)
1	1	sac	100	5.00
2	2	bolis	5000	10.00
3	3	rats	1500	45.00

Exemple 4 : Regla de negoci

- Objectiu: Una única sentència de modificació no pot augmentar la quantitat total en estoc dels productes en més d'un 50%.

```
CREATE TABLE TEMP(old_qtt integer);

CREATE FUNCTION update_items_before()RETURNS trigger AS $$
BEGIN
  INSERT INTO temp SELECT sum(qtt) FROM items;
  return null;
END
$$ LANGUAGE plpgsql;

CREATE FUNCTION update_items_after()RETURNS trigger AS $$
DECLARE
  oldqtt integer default 0;
  newqtt integer default 0;
BEGIN
  SELECT old_qtt into oldqtt FROM temp;
  DELETE FROM temp;
  SELECT sum(qtt) into newqtt FROM items;
  IF (newqtt>oldqtt*1.5) THEN
     RAISE EXCEPTION 'Violació regla de negoci';
  END IF;
  RETURN NULL;
END
$$ LANGUAGE plpgsql;

CREATE TRIGGER regla_negociA BEFORE UPDATE ON items
    FOR EACH STATEMENT EXECUTE PROCEDURE update_items_before();

CREATE TRIGGER regla_negociD AFTER UPDATE ON items
    FOR EACH STATEMENT EXECUTE PROCEDURE update_items_after();
```

Exemple 4 : Execució

Contingut inicial de la taula items

```
select * from items;
```

Panel de Salida

Salida de datos | Comentar | Mensajes | Historial

	item integer	name character(25	qtt integer
1	1	sac	100
2	2	bolis	5000
3	3	rats	500

Les accions del disparador eviten que es violi la regla de negoci, però

NO ÉS UNA BONA SOLUCIÓ!!

```
update items set qtt=qtt*3;
```

Panel de Salida

Salida de datos | Comentar | Mensajes | Historial

```
ERROR:  Violació regla de negoci

********** Error **********

ERROR: Violació regla de negoci
Estado SQL:P0001
```

Exemple 5 : Regla de negoci

- Objectiu: No pot ser que una única sentència de modificació augmenti la quantiat total en estoc dels productes en més d'un 50% → **SOLUCIÓ INCREMENTAL**

```
CREATE TABLE TEMP(
        old_qtt integer,
        incr    integer);

CREATE FUNCTION update_items_before()RETURNS trigger AS $$
BEGIN
  DELETE From temp;
  INSERT INTO temp(old_qtt,incr) select sum(qtt),0 from items;
  return null;
END
$$ LANGUAGE plpgsql;

CREATE FUNCTION update_items_inc()RETURNS trigger AS $$
DECLARE
  oldqtt integer default 0;
  suma_incr integer default 0;
BEGIN
  UPDATE temp
  SET incr=incr+(NEW.qtt-OLD.qtt);
  SELECT old_qtt,incr INTO oldqtt,suma_incr FROM temp;
  IF (suma_incr>oldqtt*0.5) THEN
     RAISE EXCEPTION 'ViolaciÛ regla de negoci';
  END IF;
  return NEW;
END
$$ LANGUAGE plpgsql;
```

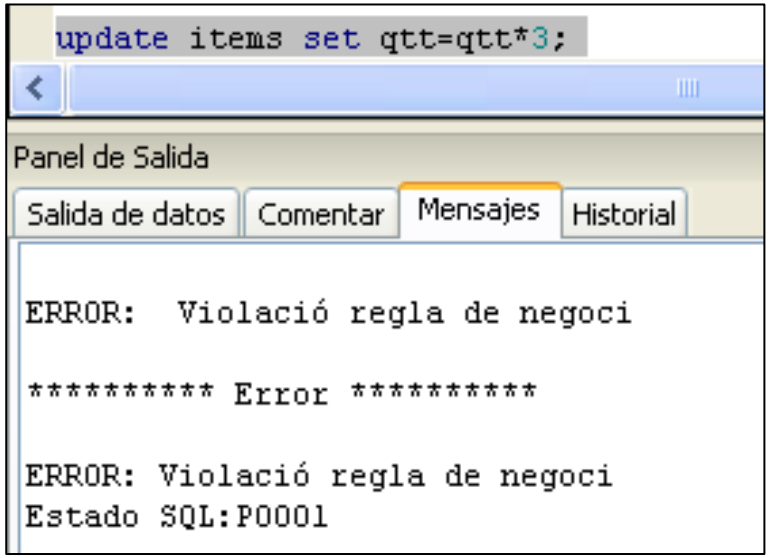
Exemple 5 : Regla de negoci

Solució Incremental

```
drop trigger regla_negociA on items;
CREATE TRIGGER regla_negociA BEFORE UPDATE ON items
    FOR EACH STATEMENT EXECUTE PROCEDURE update_items_before();

drop trigger regla_negociD on items;
CREATE TRIGGER regla_negociD BEFORE UPDATE ON items
    FOR EACH ROW EXECUTE PROCEDURE update_items_inc();
```

Exemple 5 : Execució

- Objectiu: No pot ser que una única sentència de modificació augmenti la quantiat total en estoc dels productes en més d'un 50% → SOLUCIÓ INCREMENTAL



- En postgreSQL pot haver més d'un disparador associat a un esdeveniment i tipus de trigger. En aquest cas els disparadors es disparen segons l'ordre alfabètic del nom del trigger.

Exemple 7: Auditoria

- Objectiu: Auditar els esborrats i modificacions de la taula items.

```
DROP TABLE log_record2;
CREATE TABLE log_record2(
  username char(8),
  update_time timestamp,
  operacio varchar(6));


drop function inserta_log();
CREATE FUNCTION inserta_log() RETURNS trigger AS $$
    BEGIN
         insert into log_record2 values (current_user,current_date,TG_OP);
       RETURN NULL;
    END;
$$ LANGUAGE plpgsql;


CREATE TRIGGER audit_items
AFTER UPDATE OR DELETE ON items
FOR EACH STATEMENT EXECUTE PROCEDURE inserta_log();

update items set qtt=qtt+10 where item<>3;
delete from items where item=3;
select * from log_record2;
```

anel de Salida

Salida de datos | Comentar | Mensajes | Historial

	username character(8)	update_time timestamp without	operacio character varying(6
1	bd	2008-02-04 00:00:00	UPDATE
2	bd	2008-02-04 00:00:00	DELETE

Contingut inicial de la taula Items

```
select * from items;
```

anel de Salida

Salida de datos | Comentar | Mensajes | Historial

	item integer	name character(25	qtt integer
1	1	sac	100
2	2	bolis	5000
3	3	rats	500

RELACIÓ ENTRE DISPARADORS I RESTRICCIONS

- En un disparador BEFORE, les accions es disparen abans d'executar l'operació i comprovar les restriccions
- En un disparador AFTER, les accions es disparen després d'executar l'operació i comprovar les restriccions

```
CREATE TABLE pare(
  a integer primary key);
drop table fill cascade;
CREATE TABLE fill(
  b integer references pare);


drop function inserir();
CREATE FUNCTION inserir() RETURNS trigger AS $$
BEGIN
 if ((SELECT count(*) FROM pare WHERE a=NEW.b)=0) THEN
    INSERT INTO pare VALUES (NEW.b);
 END IF;
 RETURN NEW;
END;
$$ LANGUAGE plpgsql;

drop trigger restrictl on fill;
CREATE TRIGGER restrictl BEFORE INSERT ON fill
FOR EACH ROW EXECUTE PROCEDURE inserir();

INSERT INTO fill values (1);

select * from fill;
select * from pare;
```

Panel de Salida

Salida de datos | Comentar | Mensajes | Historial

	a integer
1	1

I si el disparador s'executa AFTER?

DISPARADORS EN CASCADA

Create Trigger del_a AFTER delete on a FOR EACH ROW (Execute Procedure p1())	Create Function p1() Begin **Delete From b**…. End
Create Trigger del_b AFTER delete on b FOR EACH ROW (Execute Procedure p2())	Create Function p2() Begin **Delete From C**…. End
Create Trigger del_c AFTER delete on c FOR EACH ROW (Execute Procedure p3())	**I SI?** Create Function p3() Begin **Delete From a...** End

Consideracions de disseny

- Sovint existeixen diverses solucions vàlides per resoldre un mateix problema (veure exemples 4 i 5).
- Cal buscar la solució més eficient per cada situació. En general, cal:
  - Estalviar accessos innecessàris a la BD
  - Evitar fer feina innecessària
  - Etc.

Consideracions de disseny

Exemple:

Comprovar la restricció "*El sou d'un empleat no pot baixar*" mitjançant disparadors.

```
CREATE TABLE empleats (
        num_empl INTEGER PRIMARY KEY,
        sou INTEGER NOT NULL (CHECK sou>0),
        ….);
```

En principi, cal comprovar les restriccions el més aviat possible. Això inclou les restriccions implementades amb disparadors

En aquest cas => TRIGGER BEFORE / FOR EACH ROW

- Però si la sentència que dispara el disparador viola la restricció d'integritat CHECK sou>0

```
UPDATE empleats
SET sou=0
WHERE num_empl=10;
```

- Quan s'han de comprovar les restriccions d'integritat de la BD? Abans o després d'executar les accions del disparador?
    - Si la restricció check sou>0 es viola molt sovint, potser =>TRIGGER AFTER / FOR EACH ROW

MOLTS ASPECTES A TENIR EN COMPTE!

Consideracions de disseny

- L'efecte d'una sentència d'actualització amb la presència de disparadors depèn de l'estat de la BD. Com més gran és el nombre de disparadors, més difícil es fa saber aquest efecte.

- Seria convenient disposar d'eines d'ajuda al disseny i a la depuració dels programes, però…

- Conclusió:

  si l'acció del disparador és raise exception, cap problema;
  altrament alerta !!!

Privilegis

- Una Base de Dades té molts objectes, molts usuaris i molts grups d'usuari. No tots els usuaris han d'accedir a tots els objectes. L'SGBD ha d'establir un mecanisme de control d'accés dels usuaris sobre aquests objectes.

- Aquest mecanisme es basa en el concepte de PRIVILEGI:

  L'autorització que es dóna a un
  usuari / grup d'usuaris
  per realitzar una
  operació
  sobre un
  objecte d'un esquema

- Els privilegis s'assignen i es revoquen amb les sentències GRANT i REVOKE.

Privilegis

- SQL defineix 9 tipus de PRIVILEGIS:
  - SELECT
  - INSERT
  - UPDATE
  - DELETE

  Poden tenir associada una llista d'atributs

  Aplicables a una taula o vista
  - REFERENCES

    És el dret a fer referència a una taula en una restricció d'integritat
  - USAGE

    És el dret a utilitzar altres objectes en les pròpies declaracions
  - TRIGGER

    És el dret a definir disparadors sobre una taula
  - EXECUTE

    És el dret a executar una peça de codi, per exemple procediments
  - UNDER

    És el dret a crear subtipus d'un tipus donat
  - ALL

Privilegis

- Quan un usuari crea un esquema s'identifica amb la clàusula AUTHORIZATION i té tots els privilegis sobre ell. L'esquema serà inaccessible a altres usuaris fins que el propietari d'aquest esquema autoritzi privilegis a altres usuaris amb la sentència GRANT.

- Hi ha un authoritzation id especial: Public

- Quan una sessió es comença amb una connexió tenim l'oportunitat d'indicar l'usuari amb la clàusula USER

  CONNECT TO nom_servidor **AS** nom_connexio **USER** ident_usuari

- Per tant, podrem executar una operació SQL només si l'usuari identificat té tots els privilegis necessaris per fer l'operació sobre els objectes involucrats

GRANT i REVOKE

- Autoritzant privilegis:

  GRANT privilegis **ON** objectes **TO** usuaris [**WITH GRANT OPTION**];

- Revocant privilegis:

  REVOKE [GRANT OPTION FOR] privilegis **ON** objectes **FROM** usuaris [**CASCADE | RESTRICT**];

- Exemple:

  Júlia:

  CREATE TABLE empleats (n integer, nom char(10), sou integer);

  GRANT insert, delete, update(sou) ON empleats TO anna;

  GRANT select(n,nom) ON empleats TO anna WITH GRANT OPTION;

  GRANT select(sou) ON empleats TO anna, pere;

  GRANT update(sou) ON empleats TO joan;

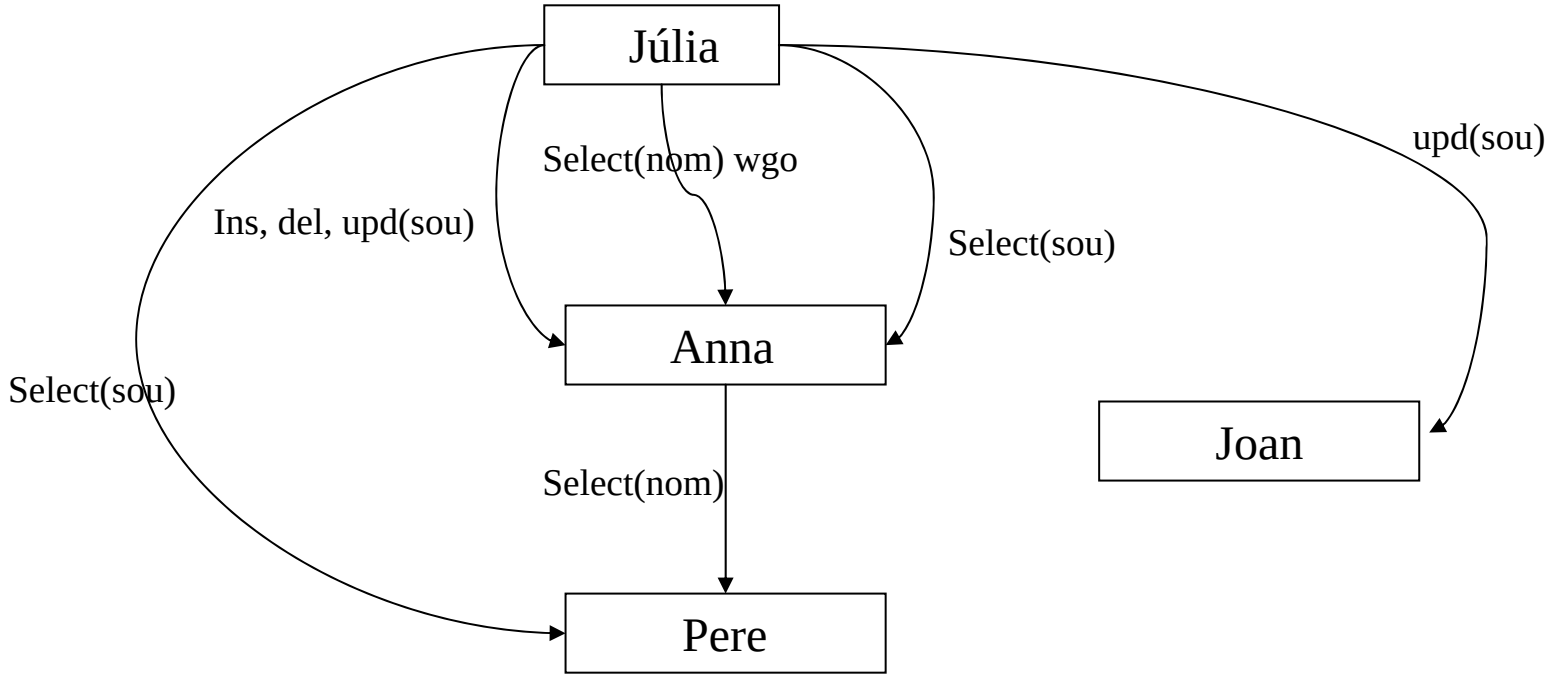
  Anna:

  GRANT select(nom) ON empleats TO pere;

  Joan:

  UPDATE empleats set sou=4;

Diagrama d'autoritzacions



Júlia: GRANT insert, delete, update(sou) ON empleats TO anna;
GRANT select(nom) ON empleats TO anna WITH GRANT OPTION;
GRANT select(sou) ON empleats TO anna, pere;
GRANT update(sou) ON empleats TO joan;
Anna: GRANT select(nom) ON empleats TO pere;
Júlia: REVOKE select(nom) ON empleats FROM anna CASCADE;

Moltes Subtileses

- Si la Júlia en lloc de GRANT select(sou) ON empleats TO pere hagués fet
  GRANT select(nom) ON empleats TO pere
  després del revoke el pere conservaria el privilegi select(nom)

- Si la Júlia en lloc de GRANT select(sou) ON empleats TO pere hagués fet
  GRANT select ON empleats TO pere
  després del revoke el pere conservaria el privilegi select

- El Joan necessita privilegis diferents per:
  UPDATE empleats set sou=4;
  UPDATE empleats set sou=sou-1;

- Júlia: GRANT select ON empleats TO toni WITH GRANT OPTION
  Toni: GRANT select ON empleats TO enric
  Júlia: REVOKE GRANT OPTION FOR select FROM empleats TO toni

Privilegis i Vistes

- Exemple:

  Donada una taula empleats (nemp, adreça, sou) es vol que l'empleat Joan només pugui veure el seu sou.

  GRANT + VISTES = precisió en les autoritzacions

```
CREATE VIEW Sou_joan AS
    SELECT sou FROM empleats WHERE nemp="joan";
GRANT SELECT ON Sou_joan TO joan;
```

- Els privilegis que es poden donar sobre una vista són els mateixos que es poden donar a una taula: SELECT, INSERT, UPDATE, DELETE

ROLS

- Nombre elevat de GRANT per implantar la BD !!
- **ROL:** és una agrupació de privilegis definida per a un grup d'usuaris específics.
  - permet al DBA estandarditzar i canviar els privilegis de molts usuaris tractant-los com a membres d'una classe
  - Similar al concepte de grup dels SO
- **DBA :**

```
CREATE ROLE lector                  CREATE ROLE escriptor
GRANT select ON T1 TO lector        GRANT update ON T1 TO escriptor
GRANT select ON T2 TO lector        GRANT update ON T2 TO escriptor
                                    GRANT lector TO escriptor
```

- **DBA o un usuari amb grant option**

```
GRANT lector TO Joan, Anna WITH GRANT OPTION
GRANT escriptor to Toni
```

- **DBA:** Manipulació dinàmica de privilegis

```
GRANT select on T3 to lector
GRANT update on T3 to  escriptor
REVOKE update on T2 from escriptor
GRANT update(a) on T2 to escriptor
```

- **Usuari**: Han d'activar els rols

```
Anna:    SET ROLE lector
```