PAU CASESNOVES
Centre Integrat de Formació Professional

Desenvolupament d'aplicacions multiplataforma

Accés a dades

UT 5.1. Object Relational Data Bases

Joan Pons Tugores

Índex de continguts

Bases de dades objecte relacionals

Les diferents versions del llenguatge SQL han anat incorporant diverses característiques que els donen cada cop més flexibilitat per treballar directament amb objectes. La principal revisió es va produir el 1999 donant lloc a l'SQL99.

Es tracta d'una revisió profunda que afegeix un nombre considerable de tipus poc convencionals, a més de permetre també la definició de tipus de dades compostos o estructurats. Els principals tipus que incorpora són:

- **Boolean**. Fins aleshores, els valors lògics se solien indicar usant el tipus BIT.
- **Grans objectes**. SQL99 defineix dos tipus d'objectes grans, l'anomenat BLOB (Binary Large Object), adequat per emmagatzemar dades binàries com ara imatges, vídeos, música, certificats digitals, etc. L'altre tipus s'anomena CLOB (Character Large Object), ideal per a dades extensives de tipus text com ara informes, pàgines web, articles, etc.
- **Col·leccions**. Permet emmagatzemar de forma directa col·leccions senceres de dades tant de tipus bàsic com de tipus estructurat.
- **Tipus compostos o estructurats**. Gràcies a la incorporació d'aquests tipus de dades és possible crear tipus de dades definits per l'usuari.
- **Referències a tipus estructurats**. Es tracta d'un tipus especial que actua com a apuntador de tipus compostos. Són útils perquè permeten fer una abstracció del lloc (taula) on realment s'emmagatzemaran aquests tipus. El sistema està preparat per realitzar un emmagatzematge per defecte, de manera que aquests tipus són l'única manera de referenciar les dades emmagatzemades.

La incorporació de tots aquests tipus de dades aporta molta més flexibilitat a l'hora de mapar les classes del model fent servir directament el llenguatge de definició DDL. A més,

SQL amplia la sintaxi del llenguatge de consulta per poder usar directament els nous tipus, de forma semblant a la manipulació dels atributs dels objectes.

PostgreSQL

PostgreSQL és una base de dades relacional de codi obert que incorpora moltes de les característiques definides per SQL99 i que d'altres sistemes gestors de bases de dades encara no han incorporat.

Algunes de les principals característiques del *PostgreSQL* són, entre altres:

- Alta concurrència: Mitjançant un sistema denominat MVCC (Accés concurrent multiversió, per les seves sigles en anglès) *PostgreSQL* permet a un procés escriure en una taula, mentre d'altres hi accedeixen, sense bloquejar-se mútuament.
- Àmplia varietat de tipus natius: *PostgreSQL* suporta nativament: Nombres de precisió arbitrària; Text de llargada il·limitada; Figures geomètriques (amb una varietat de funcions associades); Adreces IP (IPv4 i IPv6).
- Tipus definits per l'usuari.
- Funcions. Poden ser escrites en diversos llenguatges, C, Java, PL, ...

Característiques noves de PostgreSQL

Boolean

PostgreSQL suporta el tipus BOOLEAN declarat com indica l'estàndard:

```
CREATE TABLE IF NOT EXISTS proves."Clients"
(
    id integer NOT NULL,
    nom character varying(75),
    recomanaria boolean,
    CONSTRAINT "Clients_pkey" PRIMARY KEY (id)
)
```

Els valors que soporta són:

- Per Veritat: TRUE, 't', 'true', 'y', 'yes', 'on', '1'
- Per Fals: FALSE, 'f', 'false', 'n', 'no', 'off', '0'

Per exemple:

```
INSERT INTO proves."Clients"(id, nom, recomanaria) VALUES (1, 'Jo Mateix', TRUE);
```

```
update Clients set recomanaria='off' where id=1;
```

Col·leccions

Els tipus que especifiquen col·leccions presenten també algunes diferències sintàctiques. Mentre que l'estàndard admet formes com LIST, SET, MULTISET o ARRAY, *PostgreSQL* només suporta el tipus ARRAY, el qual pot expressar-se seguint la forma estàndard:

```
 CREATE TABLE IF NOT EXISTS proves."Clients"
(
    id integer NOT NULL,
    nom character varying(75),
    recomanaria boolean,
    emails character varying(75) ARRAY,
    CONSTRAINT "Clients_pkey" PRIMARY KEY (id)
)
```

O bé usant la forma específica de PostgreSQL:

```
CREATE TABLE IF NOT EXISTS proves."Clients"
(
    id integer NOT NULL,
    nom character varying(75),
    recomanaria boolean,
    emails character varying(75)[ ],
    CONSTRAINT "Clients_pkey" PRIMARY KEY (id)
)
```

Com a curiositat, a diferència de molts llenguatges de programació, la primera posició de l'array és la 1 i no la 0.

Fixeu-vos que especificant l'atribut *emails* dels clients com un tipus ARRAY ens evitarem haver de crear una taula on emmagatzemar-los.

Insert

Tenim dues maneres possibles d'especificar arrays:

```
insert into proves."Clients"
  VALUES (2, 'Un Altre', FALSE,
          ARRAY['unaltre@mail.org','unaltre@feina.com'], );
```

o de forma alternativa

```
insert into proves."Clients"
  VALUES (1, 'Jo Mateix', TRUE,
          '{"jomateix@mail.org","jomateix@feina.com"}')
```

Si tenim un array bidimensional:

```
INSERT INTO sal_emp    VALUES ('Carol',
    ARRAY[20000, 25000, 25000, 25000],
    ARRAY[['breakfast', 'consulting'], ['meeting', 'lunch']]);
```

Update

Podem actualitzar tot l'array de cop:

```
UPDATE proves."Clients"
SET emails = '{"nouEmail1@mail.com", "nouEmail2@mail.com",
          "nouEmail3@mail.com"}' WHERE nom='Jo Mateix';
```

L'array quedarà:

```
{'nouEmail1@mail.com', 'nouEmail2@mail.com', 'nouEmail3@mail.com'}
```

O només una posició. Si la posició especificada no existeix crea totes les necessàries inicialitzades a null i actualitza la demanada:

```
UPDATE proves."Clients" SET emails[5] = 'nouEmail5@mail.com'
        WHERE nom='Jo Mateix';
```

L'array quedarà:

```
{'nouEmail1@mail.com', 'nouEmail2@mail.com', 'nouEmail3@mail.com',
        NULL, 'nouEmail5@mail.com'}
```

Podem actualitzar un rang de posicions de l'array:

```
UPDATE proves."Clients" SET emails[1:2] =
        '{"rang1@mail.org","rang2@mail.org"}'  WHERE id=1;
```

Obtindrem:

```
{'rang1@mail.org', 'rang2@mail.org', 'nouEmail3@mail.com', NULL,
'nouEmail5@mail.com'}
```

Afegir un valor al final:

```
update proves."Clients" set emails=array_append(emails,'afegit@mail.org');
```

Quedarà:

```
{'rang1@mail.org', 'rang2@mail.org', 'nouEmail3@mail.com', NULL,
'nouEmail5@mail.com', 'afegit@mail.org'}
```

Eliminar un valor de l'array. Elimina totes les aparicions d'aquest valor. L'array disminueix la seva longitud:

```
UPDATE proves."Clients" SET emails = array_remove(emails, 'rang2@mail.org') WHERE id=1;
```

L'array haurà disminuït la seva longitud:

```
{'rang1@mail.org', 'nouEmail3@mail.com', NULL, 'nouEmail5@mail.com', 'afegit@mail.org'}
```

Més informació sobre operadors i funcions a l'enllaç:

https://www.postgresql.org/docs/17/functions-array.html.

Select

Podem seleccionar una posició de l'array:

```
SELECT emails[2] FROM proves."Clients" where id=1;
```

Un rang de posicions de l'array,

```
SELECT emails[2:4] FROM proves."Clients" where id=1;
```

encara que sigui multidimensional:

```
SELECT schedule[1:2][1:1] FROM sal_emp WHERE name = 'Bill';
```

Podem consultar el nombre de dimensions de l'array:

```
SELECT array_ndims(emails) FROM proves."Clients" where id=1;
```

Podem consultar la longitud d'una de les dimensions de l'array:

```
SELECT array_length(emails,1) FROM proves."Clients" where id=1;
```

Where

Podem filtrar per una posició:

```
SELECT * FROM proves."Clients" where emails[1]='rang1@mail.org'
```

```
SELECT * FROM proves."Clients" where emails[1] <> 'rang1@mail.org'
```

Podem comprovar que un valor coincideixi amb qualsevol valor de l'array:

```
SELECT * FROM proves."Clients" where 'rang1@mail.org' = ANY (emails)
```

Podem comparar un valor amb tots els de l'array:

```
SELECT * FROM proves."Clients" where 'rang1@mail.org' <> ALL (emails)
```

Tipus composts

DDL

L'estàndard admet dues formes: l'ús de la paraula clau ROW amb la composició de camps tancada entre parèntesis o bé la declaració prèvia del tipus compost que volem utilitzar.

PostgreSQL suporta només el darrer, la declaració prèvia del tipus compost.

Per exemple cream el tipus *t_adreca*, amb l'adreça (carrer, número, pis, ...), el codi postal, la població i el país.

```
CREATE TYPE proves.t_adreca AS
(
	adreca character varying(255),
	codipostal character varying(5)[],
	poblacio character varying(100)[],
	pais character varying(100)[]
);
```

I el tipus *t_telefon,* amb un camp descriptiu per saber si és el personal, de la feina, ...

```
CREATE TYPE proves."t_telefon" AS(
	tipus character varying(25),
	numero character varying(20)
);
```

I els utilitzam com a tipus de les columnes de la taula:

```
CREATE TABLE IF NOT EXISTS proves."Clients"
(
    id integer NOT NULL,
    nom character varying(75) COLLATE pg_catalog."default",
    emails character varying(75)[] COLLATE pg_catalog."default",
    recomanaria boolean,
    adreca proves.t_adreca,
    telefon proves.t_telefon,
    CONSTRAINT "Clients_pkey" PRIMARY KEY (id)
)
```

Els tipus compostos es declaren com si fossin taules però no s'indiquen restriccions, només s'accepten els noms i els tipus de cada camp. Fixeu-vos que fins i tot podem fer servir un tipus compost en una col·lecció.

És **important** aclarir que les taules creen, per defecte, un tipus compost amb el mateix nom de la taula. Cal anar en compte de no posar als tipus que haguem de definir el nom d'una taula existent, perquè ens donaria un error de duplicitat de nom.

Extrapolant aquesta consideració, el fet que les taules creïn tipus per defecte ens permet usar el nom de qualsevol taula com a tipus definit. Així, per exemple, si tenim definides la taula *TipusClient* podem usar el seus nom en la definició d'altres taules:

```
CREATE TABLE IF NOT EXISTS proves."TipusClient"
(
    id integer NOT NULL,
    nom character varying[] COLLATE pg_catalog."default",
    CONSTRAINT "Clients_pkey2" PRIMARY KEY (id)
)
```

Llavors podem crear la taula client com:

```
CREATE TABLE IF NOT EXISTS proves."Clients"
(
    id integer NOT NULL,
    nom character varying(75) COLLATE pg_catalog."default",
    emails character varying(75)[] COLLATE pg_catalog."default",
    recomanaria boolean,
    adreca proves.t_adreca,
    telefon proves.t_telefon,
    tipus proves."TipusClient",
    CONSTRAINT "Clients_pkey" PRIMARY KEY (id)
)
```

DML

En instruccions DML els objectes es creen utilitzant ROW

```
INSERT INTO Client(... , telefon) VALUES (..., ROW('Casa', 874563254));
```

En consultes es poden accedir directament els atributs de l'objecte

```
Select (telefon).tipus from Client where (telefon).nombre=874563254
```

En accedir a un atribut del tipus complexe s'ha de posar el nom del camp entre parèntesis, si no *PostgreSQL* cercaria una taula amb el nom del camp.

Es poden modificar els objectes complets o només alguns atributs individuals de l'objecte complex.

```
update client
        set telefon.numero=874563252
        where (telefon).tipus='Casa' and client.id=3;
```