Características Objeto Relacional do PostgreSQL

Prof. Fabiano Baldo

PostgreSQL

- Derivado do POSTGRES
  - Desenvolvido em Berkeley por Mike Stone-braker e seus estudantes, iniciado em 1986
- Postgres95
  - Andrew Yu e Jolly Chen adaptaram o POSTGRES para SQL e melhoraram substancialmente o código base
- PostgreSQL
  - O nome mudou em 1996, e desde então o sistema tem sido expandido para suportar várias características do SQL-92 e SQL-99

Classes PostgreSQL (1/2)

- Em Postgres uma classe representa uma coleção de instâncias de objetos de um mesmo tipo.
- Cada instância tem a mesma coleção de atributos nomeados, e cada atributo é de um tipo específico.
- Cada instância tem uma identidade de objeto permanente (OID) que é único durante sua existência.

Classes PostgreSQL (1/2)

- Pelo fato da sintaxe SQL se referir a tabelas, os termos tabelas e classes serão usados com sinônimos.
- Da mesma forma, uma tupla em SQL será uma instância e uma coluna em SQL será um atributo.

Criando uma tabela (Classe)

- Pode-se criar uma nova tabela especificando seu nome, além de todos os seus campos e respectivos tipos :

```
CREATE TABLE clima (
    cidade          varchar(80),
    temp_lo         int,          -- temperatura baixa
    temp_hi         int,          -- temperatura alta
    prec            real,         -- precipitação
    data            date
);
```


PostgreSQL

- Postgres pode ser personalizado através da especificação de tipos de dados definidos pelo usuário.
- Assim, o comando CREATE do Postgres se parece exatamente com o comando usado para criar uma tabela em um BD relacional tradicional.
- Entretanto, será visto mais adiante que as tabelas têm certas propriedades que são extensões do modelo relacional.

PostgreSQL

- Todos os comandos SQL usuais para a criação, recuperação e modificação de tabelas continuam disponíveis.
- Entretanto, há a adição de:
  - Herança
  - Valores não atômicos
  - Tipos Compostos
  - Funções e operações definidas pelo usuário

Herança

```
CREATE TABLE cidades (
    nome            text,
    populacao       float,
    altitude        int     -- (em pés)
);

CREATE TABLE capitais (
    estado          char(2)
) INHERITS (cidades);
```

Herança

```
ray=# create table cidades (nome varchar(50), populacao float,
      altitude int);

ray=# \d cidades
             Table "public.cities"
   Column   |         Type          | Modifiers
------------+-----------------------+-----------
 nome       | character varying(50) |
 populacao  | double precision      |
 altitude   | integer               |

ray=# create table capitais (estado char(2)) inherits (cidades);

ray=# \d capitais
            Table "public.capitals"
   Column   |         Type          | Modifiers
------------+-----------------------+-----------
 nome       | character varying(50) |
 populacao  | double precision      |
 altitude   | integer               |
 estado     | character(2)          |
Inherits: cidades
```

9

Herança

- Em Postgres, uma tabela pode herdar de uma ou mais tabelas.
- Uma consulta pode recuperar:
  - Todas as tuplas de uma tabela
  - Ou todas as tuplas de uma tabela mais todas as tuplas dos seus descendentes

Herança

- Por exemplo, a consulta seguinte recupera todas as cidades, incluindo as capitais dos estados, que estão localizadas a uma altitude superior a 500 pés:

```
SELECT nome, altitude
    FROM cidades
    WHERE altitude > 500;
```

```
|nome          | altitude      |
+--------------+-----------+
|Las Vegas     | 2174          |
|Mariposa      | 1953          |
|Madison       | 845   |
```


Herança

- Por outro lado, para recuperar todas as cidades que não são capitais de estados e que estão a mais de 500 pés de altitude, a consulta é:

```
SELECT nome, altitude
   FROM ONLY cidades
    WHERE altitude > 500;
```

```
|nome          | altitude      |
+--------------+-----------+
|Las Vegas | 2174          |
|Mariposa      | 1953          |
```

Herança

- A palavra “ONLY" antes de *cidades* indica que a consulta deve ser efetuada apenas sobre a tabela *cidades*, sem incluir as tabelas abaixo de *cidades* na hierarquia de herança.
- Vários comandos de PostgreSQL (SELECT, UPDATE e DELETE) permitem a utilização da notação ONLY

Valores Não Atômicos

- Uma das premissas do modelo relacional é que os atributos de uma relação são atômicos
  - I.e. apenas um valor único para uma dada linha e coluna
- PostgreSQL não tem essa restrição, pois atributos podem conter múltiplos valores
  - Exemplos incluem arrays e outros tipos de dados complexos.

Valores Não Atômicos - Arrays

- Postgres permite que colunas de uma tabela sejam definidas como matrizes multidimen-sionais de comprimento variável.
- Podem serem criadas matrizes de qualquer tipo, nativo ou definido pelo usuário.

```
CREATE TABLE SAL_EMP (
    nome                text,
    pagto_semanal       int[],
    agenda              text[][]
);
```


Valores Não Atômicos - Arrays

- O tipo array é definido através dos colchetes “[ ]”
- O comando SQL anterior cria uma tabela chamada SAL_EMP com uma string (nome), um array unidimensional de int4 (pagto_semanal), que representa o salário do empregado por semana, e um array bidimensional de string (agenda), que representa a agenda semanal do empregado.

Inserindo em Arrays

- Note que os valores dos elementos devem ser delimitados por chaves “{ }” e separados por vírgulas “,”

```
INSERT INTO SAL_EMP
   VALUES ('Bill',
   '{10000, 10000, 10000, 10000}',
   '{{“reunião", “almoço"}, {}}');

INSERT INTO SAL_EMP
   VALUES ('Carol',
   '{20000, 25000, 25000, 25000}',
   '{{“aula", "consulta"}, {“reunião"}}');
```


Consultando Arrays

- Esta consulta recupera os nomes dos empregados cujo pagamento mudou na segunda semana :

```
SELECT nome
   FROM SAL_EMP
   WHERE SAL_EMP.pagto_semanal[1] <>
   SAL_EMP. pagto_semanal[2];
|nome  |
+--------+
|Carol   |
```

Consultando Arrays

- Esta consulta retorna o pagamento da terceira semana de todos os empregados:

```
SELECT SAL_EMP.pagto_semanal[3] FROM
   SAL_EMP;

|pagto_semanal |
+--------------------+
|10000               |
|25000               |
```


Consultando Arrays

- Pode-se também acessar faixas retangulares arbitrárias da matriz, ou submatrizes.
- Uma faixa da matriz é especificada escrevendo
  - [*limite_inferior:limite_superior*] para uma ou mais dimensões da matriz
- A consulta mostra o primeiro item da agenda do Bill para os dois primeiros dias da semana

```
SELECT SAL_EMP.agenda[1:2][1:1]
   FROM SAL_EMP
   WHERE SAL_EMP.nome = 'Bill';
|agenda                              |
+------------------------------------+
|{{“reunião"},{“treinamento"}} |
```

Tipos Compostos

- Um tipo composto descreve a estrutura de uma linha ou registro;
- Ele define uma lista de nomes de campos com seus respectivos tipos de dados.
- Um tipo composto pode ser utilizado para definir o tipo de um campo de uma tabela

Criando um Tipo Composto

- CREATE TYPE permite ao usuário definir um novo tipo para o uso no banco de dados.
- O usuário que define um tipo se torna seu dono.
- O nome do novo tipo deve ser único dentro dos tipos definidos para um banco de dados.

```
CREATE TYPE catalogo AS (
    nome            text,
    id_fornecedor   integer,
    preco           numeric
);
```

Usando um Tipo Composto

- Após definir os tipos, estes podem ser utilizados para criar tabelas:

```
CREATE TABLE estoque (
    item            catalogo,
    quantidade      integer
);
```

- Inserindo dados na tabela estoque

```
INSERT INTO estoque VALUES (ROW('Arroz',
    45, 5.40), 100);
```


Usando um Tipo Composto

- Também pode-se definir campos não atômicos de tipos compostos:

```
CREATE TABLE estoques (
    item            catalogo[],
    quantidade      integer[],
    cd_estoque      integer
);
```

Inserindo Valores Composto

- Forma geral: '(*val1, val2, ...*)'
- Os valores devem ser envolvidos por aspas e parênteses, e separados por vírgula.
  - Exemplo: **'("Arroz", 45, 5.40), 100)';**
- Também pode se utilizada a expressão ROW para construir valores compostos:
  - Exemplo: **(ROW('Arroz', 45, 5.40), 100)**

Acessando Valores Compostos

- Para acessar um campo de uma coluna composta deve ser escrito:
  - **(<nome_coluna>).<nome_campo>**
- É similar a seleção de um campo de uma tabela.
- Exemplo:

```
SELECT (item).nome FROM estoque WHERE
  (item).preco > 9.99;
```

Funções Definidas pelo Usuário

- CREATE FUNCTION permite a um usuário do Postgres criar uma função em uma BD.
- Esse usuário é considerado o *dono* da função.

```
CREATE FUNCTION nome ( [ ftipo [, ...] ] )
   RETURNS rtipo
   AS {definição_SQL}
   LANGUAGE 'nome_ling'
   [ WITH ( atributo [, ...] ) ]

CREATE FUNCTION nome ( [ ftipo [, ...] ] )
   RETURNS rtipo
   AS arquivo_objeto, link_simbolico
   LANGUAGE 'C'
   [ WITH ( atribute [, ...] ) ]
```


Função SQL Simples

- CREATE FUNCTION one() RETURNS int4
  AS 'SELECT 1 AS RESULTADO'
  LANGUAGE 'sql';

SELECT one() AS resposta;

```
resposta
--------
       1
```

Função mais complexa

- Considere a seguinte função que pode ser utilizada para debitar uma conta bancária:

```
CREATE FUNCTION TP1 (int4, float8) RETURNS float8
AS ‘update BANCO set saldo = BANCO.saldo - $2
   where BANCO.nuconta = $1;
   select saldo from banco
    where nuconta = $1; ’ language 'sql';
```

- Um usuário pode executar essa função para debitar a conta 17 em $100.00 como segue:

```
select (x = TP1( 17,100.0));
```


Funções SQL em Tipos Compostos

- Quando criando uma função com tipos compostos, deve-se incluir os atributos desse argumento. Se EMP é uma tabela contendo dados de empregados, uma função para dobrar o salário deve ser:

```
CREATE FUNCTION dobrar_salario(EMP) RETURNS integer
    AS ' SELECT $1.salario * 2 AS salario; ' LANGUAGE ‘SQL’;

SELECT nome, dobrar_salario(EMP) AS sonho FROM EMP WHERE
    EMP.nome = 'Sam';
name | sonho
------+-------
Sam | 2400
```

Note o uso da sintaxe $1.salário para selecionar o valor de um campo do argumento linha. Também note como a chamada do comando SELECT usa um nome de tabela para denotar toda a linha atual da tabela como um valor composto.

Funções SQL em Tipos Compostos

- Também é possível criar uma função que retorne um tipo composto. Esse é um exemplo de uma função que retorna uma única linha de EMP:

```
CREATE FUNCTION novo_emp() RETURNS EMP
  AS ' SELECT text "None" AS nome,
  1000 AS salario,
  25 AS idade; ' LANGUAGE 'SQL';
```


Funções Externas

- Esse exemplo cria uma função C pela chamada da rotina de uma biblioteca criada por um usuário. Essa rotina calcula um dígito verificador e retorna TRUE se o parâmetro está correto. Ela deve ser usada em uma restrição de verificação.

```
CREATE FUNCTION ean_checkdigito(bpchar, bpchar) RETURNS
   bool
  AS '/usr1/proj/bray/sql/funcs.so' LANGUAGE 'c';
CREATE TABLE produto (
  id       char(8) PRIMARY KEY,
  eanprefix char(8) CHECK (eanprefix ~ '[0-9]{2} [0-9]{5}')
              REFERENCES brandname(ean_prefix),
  eancode   char(6) CHECK (eancode ~ '[0-9]{6}'),
  CONSTRAINT ean    CHECK (ean_checkdigito(eanprefix,
   eancode)));
```