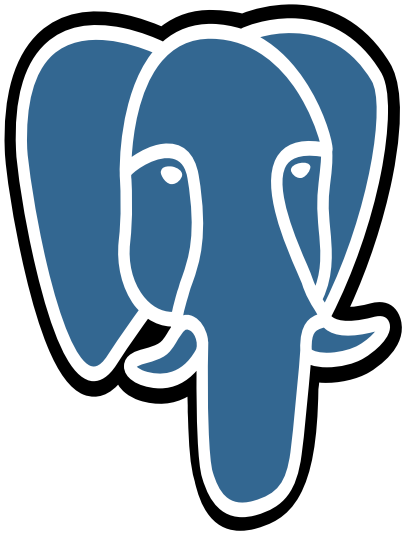
Introducción a la base de datos relacional PostgreSQL

Publicado por pico.dev el , actualizado el .
blog-stack gnu-linux planeta-codigo planeta-linux programacion
Enlace permanente Comentarios

De todas las funcionalidades que tiene SQL muchos desarrolladores solo usamos un pequeño conjunto de las posibilidades del lenguaje. Algunas bases de datos no implementan muchas posibilidades del lenguaje SQL y no son usables en esos sistemas, PostgreSQL es una de las bases de datos relacionales que mejor soporta el estándar ANSI-SQL. Conociendo sus posibilidades podremos implementar funcionalidades de forma más sencilla o con mejor rendimiento.



Aún con el reciente auge de las bases de datos NoSQL las bases de datos relacionales siguen siendo la opción usada mayoritariamente para persistir los datos de una aplicación. El potente lenguaje SQL permite obtener, modificar, insertar y eliminar datos de forma declarativa. Una característica deseada de las bases de datos relacionales es la de mantener con transacciones la integridad referencial y consistencia de los datos en todo momento que las bases de datos NoSQL no ofrecen aunque estas últimas a cambio ofrecen mejores opciones para escalar. Por otro lado los datos ya sean en una base de datos relacional o NoSQL seguirán un esquema aunque en este último caso no se exija, las bases de datos relacionales al exigir que los datos sigan un esquema evitará inconsistencias y los tipos de los datos serán los definidos en la tabla de datos en las que se guarden.

De las bases de datos relacionales más utilizadas que tienen una licencia de software libre están MySQL, MariaDB y PostgreSQL, con licencia privativa y comerciales están Microsoft SQL Server, Oracle y DB2 siendo su coste significativo en algunos casos solo alcanzable por grandes organizaciones. PostgreSQL con su licencia de software libre es una de las bases de datos más avanzadas soportando muchas de las opciones definidas en el estándar del lenguaje SQL.

Muchos desarrolladores conocemos las opciones básicas del lenguaje SQL, las sentencias *insert*, *update*, *delete* y *select*, sin embargo las últimas versiones del lenguaje SQL añade muchas posibilidades que quizá desconozcamos. PostgreSQL por ejemplo soporta inserciones de múltiples filas en una misma sentencia, actualización o inserción con la sentencia *upsert*, *window functions*, *common table expressions* o consultas recursivas. PostgreSQL además tiene un sistema de tipos avanzado pudiendo definir tipos de datos personalizados y funciones sobre esos tipos asi como herencia que son motivos por los cuales se autodenomina una base de datos *object-relational*.

Veamos algunos ejemplos de estas características del lenguaje SQL y que PostgreSQL soporta siendo una de las bases de datos relacionales más *ANSI-SQL compliant*. Para los casos demostrativos de las sentencias SQL usaré una base de datos de ejemplo con unas pocas tablas y datos sobre ciudades, países, población y lenguajes obtenida de PgFoundry Sample Databases, hay varias en concreto usaré la base de datos *world*. En la página de bases de datos ejemplo para PostgreSQL hay otras.

Para una fácil instalación de una instancia de la base de datos PostgreSQL usaré Docker con la que una vez terminados los ejemplos se puede eliminar sin dejar ningún rastro. Si aún no has usado Docker puedes leer la serie de artículos sobre Docker que he escrito.

Instalación PostgreSQL con Docker

Una vez instalado Docker e iniciado su servicio y con el comando *docker-compose* y el archivo *docker-compose.yml* que contiene la definición del contenedor lo iniciamos con el comando *docker-compose up*. El comando *docker ps* lista los contenedores en ejecución y con el comando *docker exec* iniciamos un proceso bash en el contenedor indicado con su identificativo.

Comandos básicos del *shell* psql

El *shell* de psql usa varios comandos precedidos por una contrabarra para interpretar algunos comandos muy útiles como listar las bases de datos, cambiar de base de datos de trabajo, listar las tablas de una base de datos, mostrar la definición de una tabla para saber sus campos y tipos o salir del *shell*. Los siguientes son solo unos pocos de los disponibles.

- *l*: lista las bases de datos de la instancia de PostgreSQL.
- *connect [database]*: cambia de base de datos actual de trabajo.
- *dt*: lista las tablas de la base de datos actual de trabajo.
- *d+ [table]*: muestra la definición de una tabla de la base de datos actual de trabajo.
- *q*: sale del intérprete de comandos del *shell* de PostgreSQL.

- psql *shell*

Importación base de datos de ejemplo

Antes de lanzar sentencias SQL hay que crear una base de datos con varias tablas y datos, en este caso usando una base de datos de ejemplo que se descarga con el comando *wget*, se descomprime, se crea un nuevo esquema y se importan las tablas y datos, finalmente se listas las definiciones de las tablas.

```
bash-4.3# wget http://pgfoundry.org/frs/download.php/527/world-1.0.tar.gz
bash-4.3# tar -zxvf world-1.0.tar.gz

bash-4.3# psql -U admin
admin=# create database world;
admin=# \q

bash-4.3# psql -U admin world < dbsamples-0.1/world/world.sql

bash-4.3# psql -U admin
admin=# \l
List of databases


Name | Owner | Encoding | Collate | Ctype | Access privileges
-----------+----------+----------+------------+------------
+-----------------------
admin | postgres | UTF8 | en_US.utf8 | en_US.utf8 |
companies | admin | UTF8 | en_US.utf8 | en_US.utf8 |
postgres | postgres | UTF8 | en_US.utf8 | en_US.utf8 |
template0 | postgres | UTF8 | en_US.utf8 | en_US.utf8 | =c/postgres +
| | | | | postgres=CTc/postgres
template1 | postgres | UTF8 | en_US.utf8 | en_US.utf8 | =c/postgres +
| | | | | postgres=CTc/postgres
world | admin | UTF8 | en_US.utf8 | en_US.utf8 |
(6 rows)


admin=# \connect world
You are now connected to database "world" as user "admin".
world=# \dt
List of relations
Schema | Name | Type | Owner
--------+-----------------+-------
+-------
public | city | table | admin
public | country | table | admin
public | countrylanguage | table | admin
(3 rows)
world=# \d+ city
Table "public.city"
Column | Type | Modifiers | Storage | Stats target | Description
-------------+--------------+-----------+----------+--------------
+-------------
```

id | integer | not null | plain | |
name | text | not null | extended | |
countrycode | character(3) | not null | extended | |
district | text | not null | extended | |
population | integer | not null | plain | |
Indexes:
"city_pkey" PRIMARY KEY, btree (id)
Referenced by:
TABLE "country" CONSTRAINT "country_capital_fkey" FOREIGN KEY (capital) REFERENCES city(id)

world=# \d+ country
Table "public.country"
Column | Type | Modifiers | Storage | Stats target | Description
---+---+---+---+---
code | character(3) | not null | extended | |
name | text | not null | extended | |
continent | text | not null | extended | |
region | text | not null | extended | |
surfacearea | real | not null | plain | |
indepyear | smallint | | plain | |
population | integer | not null | plain | |
lifeexpectancy | real | | plain | |
gnp | numeric(10,2) | | main | |
gnpold | numeric(10,2) | | main | |
localname | text | not null | extended | |
governmentform | text | not null | extended | |
headofstate | text | | extended | |
capital | integer | | plain | |
code2 | character(2) | not null | extended | |
Indexes:
"country_pkey" PRIMARY KEY, btree (code)
Check constraints:
"country_continent_check" CHECK (continent = 'Asia'::text OR continent = 'Europe'::text OR continent = 'North America'::text OR continent = 'Africa'::text OR continent = 'Oceania'::text OR continent = 'Antarctica'::text OR continent = 'South America'::text)
Foreign-key constraints:
"country_capital_fkey" FOREIGN KEY (capital) REFERENCES city(id)
Referenced by:
TABLE "countrylanguage" CONSTRAINT "countrylanguage_countrycode_fkey" FOREIGN KEY (countrycode) REFERENCES country(code)

```
world=# \d+ countrylanguage
Table "public.countrylanguage"
Column | Type | Modifiers | Storage | Stats target | Description
-------------+--------------+-----------+----------+--------------
+-------------
countrycode | character(3) | not null | extended | |
language | text | not null | extended | |
isofficial | boolean | not null | plain | |
percentage | real | not null | plain | |
Indexes:
"countrylanguage_pkey" PRIMARY KEY, btree (countrycode, language)
Foreign-key constraints:
"countrylanguage_countrycode_fkey" FOREIGN KEY (countrycode) REFERENCES country(code)
```

view raw database-world.sh hosted with ❤ by GitHub

Para probar que la base de datos se ha importado correctamente la siguiente sentencia SQL lista el número de ciudades por país ordenados alfabéticamente o por número de ciudades descendentemente.

```
world=#
select co.name, count(ci.id) as n from country co inner join city ci on ci.countrycode = co.code group by co.code order by co.name;
name | n
-------------------------------------
+-----
Afghanistan | 4
Albania | 1
Algeria | 18
American Samoa | 2
Andorra | 1
Angola | 5
Anguilla | 2
Antigua and Barbuda | 1
Argentina | 57
Armenia | 3
Aruba | 1
Australia | 14
Austria | 6
```

Azerbaijan | 4
--More--

select co.name, count(ci.id) as n from country co inner join city ci on ci.countrycode = co.code group by co.code order by n desc;
name | n

+-----
China | 363
India | 341
United States | 274
Brazil | 250
Japan | 248
Russian Federation | 189
Mexico | 173
Philippines | 136
Germany | 93
Indonesia | 85
United Kingdom | 81
South Korea | 70
Iran | 67
Nigeria | 64
Turkey | 62
Pakistan | 59
Spain | 59
Italy | 58

--More--
view raw sample-world.sql hosted with ❤ by GitHub
Para algunas sentencias usaré una base de datos un poco más sencilla que con una tabla para almacenar empresas.

bash-4.3# psql -U admin

admin=# create database companies;

admin=# \connect companies
companies=#

create table company(
ID BIGSERIAL PRIMARY KEY NOT NULL,

```
NAME CHAR(50) UNIQUE NOT NULL,
FOUNDATION DATE NOT NULL,
ADDRESS TEXT,
EMPLOYEES INTEGER


);
```
view raw database-companies.sh hosted with ❤ by GitHub

Inserción múltiple

Si insertamos muchos datos en una misma tabla podemos insertarlos en una única sentencia en vez de múltiples para un mejor rendimiento, evitando enviar al servidor mútiples sentencias individuales.

```
companies=#

INSERT INTO company (name, foundation, address, employees) VALUES
('Oracle', '1977-06-16', 'Redwood Shores, California, Estados Unidos', 105000),
('Apple', '1976-04-01', '1 Infinite Loop, Cupertino, California, Estados Unidos', 80000),
('Microsoft', '1975-04-04', 'Redmond, Washington, Estados Unidos', 120584),
('RedHat', '1993-01-01', 'Raleigh, Carolina del Norte 100 East Davie Street', 6100),
('Sun Microsystems', '1982-02-24', '4150 Network Circle, Santa Clara, California, Estados Unidos',
35000);


SELECT * FROM company;
```

- Inserting Data

UPSERT

En algún caso quizá tengamos la necesidad de hacer un *insert* y si el registro ya existe hacer un *update*. Usando la expresión *ON CONFLICT UPDATE* conocida como *UPSERT* podemos hacer esta operación que nos evitará hacerlo de forma programática en la aplicación.

En el ejemplo, se hace una *insert* de la empresa *Canonical*, en el segundo caso como esta empresa ya está creada y hay una restricción en el nombre para que sea único se realiza un *update* y se actualiza su número de empleados pero no se inserta un nuevo registro duplicado.

```
companies=#

INSERT INTO company (name, foundation, address, employees) VALUES ('Canonical', '2004-03-05',
'London, United Kingdom', '700') ON CONFLICT (name) DO UPDATE
SET foundation = EXCLUDED.foundation, address = EXCLUDED.address, employees =
EXCLUDED.employees;
```

SELECT * FROM company;
id | name | foundation | address | employees
----+----------+------------+----------------------+-----------
2 | Canonical | 2004-03-05 | London, United Kingdom | 700
(1 row)

INSERT INTO company (name, foundation, address, employees) VALUES ('Canonical', '2004-03-05', 'London, United Kingdom', '750') ON CONFLICT (name) DO UPDATE
SET foundation = EXCLUDED.foundation, address = EXCLUDED.address, employees = EXCLUDED.employees;

SELECT * FROM company;
id | name | foundation | address | employees
----+----------+------------+----------------------+-----------
2 | Canonical | 2004-03-05 | London, United Kingdom | 750

(1 row)

- UPSERT

Common table expressions

Las cláusula *WITH* que define las *common table expressions* o *CTE* proporcionan una forma de escribir sentencias auxiliares para su uso en una sentencia más grande. Cada sentencia auxiliar de una cláusula *WITH* puede ser un *SELECT*, *INSERT*, *UPDATE* o *DELETE* y la sentencia primaria asociada a la cláusula *WITH* también puede ser un *SELECT*, *INSERT*, *UPDATE* o *DELETE*.

- WITH Queries (Common Table Expressions)

Window functions

Las *window functions* realizan cálculos sobre un conjunto de datos que están relacionados de alguna forma con la fila actual. Al contrario que las funciones de agregación el cálculo de las *window functions* no causan que las filas se agrupen en una única fila manteniéndose como filas separadas.

Usando la base de datos *world* que contienen ciudades y países con sus poblaciones con la siguiente consulta SQL se obtienen las tres ciudades más pobladas de Alemania, España, Francia e Italia con su porcentaje respecto al total del país. En este caso Berlín es la ciudad más poblada de Alemania con aproximadamente el 12% de la población de ese país. En este caso además de usar *windows functions* se usa una *Common Table Expressions* con la cláusula *WITH*.

```
WITH city_rank AS (
SELECT c.*, rank() OVER w AS rank, sum(population) OVER w AS country_cities_population
FROM city c
WHERE countrycode in ('DEU', 'ESP', 'FRA', 'ITA')
WINDOW w AS (PARTITION BY countrycode ORDER BY population DESC RANGE BETWEEN
UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING)
)
SELECT countrycode, name, CAST(population * 100 AS FLOAT) / country_cities_population AS
percentage
FROM city_rank
WHERE rank <= 3
ORDER BY countrycode, percentage DESC;
countrycode | name | percentage
-------------+------------------
+------------------
DEU | Berlin | 12.9038090097256
DEU | Hamburg | 6.49534626586983
DEU | Munich [München] | 4.55148796461471
ESP | Madrid | 17.2716981012094
ESP | Barcelona | 9.0193410129311
ESP | Valencia | 4.43580068592419
FRA | Paris | 22.9893166678457
FRA | Marseille | 8.63681668244903
FRA | Lyon | 4.81856551586274
ITA | Roma | 17.5222222494716
ITA | Milano | 8.62315477961551
ITA | Napoli | 6.64557392020253


(12 rows)
```

- Window Functions
- Window Function Calls
- Window Functions Functions
- Window Function Processing

Consultas recursivas

El modificador *RECURSIVE* cambia la sentencia *WITH* de una conveniencia sintáctica en una funcionalidad que proporciona algo que no sería posible con el SQL que soporta algunas otras bases de datos. Usando *RECURSIVE*, una cláusula *WITH* puede referenciar su propia salida. Con esta cláusula las relaciones jerárquicas pueden implementarse sin usar otras soluciones más complejas.

- WITH Queries (Common Table Expressions)

Tipo array, enumerado

Con la ayuda de los arrays podemos definir una columna con un conjunto de valores que en casos simples nos evitarán crear una tabla con una relación 1 a N. Además, con las funciones asociadas a los arrays podemos definir una columna con un conjunto de valores cuyos valores no se repitan o si la lista es un conjunto limitados de valores con un enumerado.

- Arrays
- Array Functions and Operators
- Data Types
- Enumerated Types
- Range Types

Tipo personalizado

En PostgresSQL se pueden definir nuevos tipos de datos así como nuevas funciones sobre estos tipos de datos. Una vez definidos las columnas de las tablas pueden hacer uso de ellos. Pueden ser:

- Compuestos: están formados por una lista de nombres de atributos y tipos.
- Enumerados: son una lista de una o más etiquetas.
- Rangos
- Base
- Arrays: las columnas de una tabla se pueden definir como un array multidimensional de longitud variable. Se pueden crear arrays de cualquier de los tipos incorporados por defecto y de los tipos base, enumerados y compuestos.

Creando tipos de datos personalizados se evita crear en las tablas varios campos de tipos básicos individuales pero relacionados y estos tipos se pueden reutilizar en la definición de varias tablas.

```
companies=#

CREATE TYPE company_type AS ENUM ('startup', 'pyme', 'multinational');

CREATE TABLE company (
ID BIGSERIAL PRIMARY KEY NOT NULL,
NAME VARCHAR(50) UNIQUE NOT NULL,
TYPE COMPANY_TYPE NOT NULL,
FOUNDATION DATE NOT NULL,
ADDRESS TEXT,
EMPLOYEES INTEGER CONSTRAINT employees_positive CHECK (EMPLOYEES > 0)
```

- Monetary Types
- CREATE TYPE
- CREATE FUNCTION
- Query Language (SQL) Functions
- Arrays

Índices

Los índices cuando son utilizados son una forma que mejora enormemente el rendimiento de una consulta. Permiten buscar y obtener filas específicas mucho más rápido que sin un usar un índice.

- Indexes

Índice parcial

Un índice parcial es un índice construido sobre un subconjunto de una tabla, el subconjunto es definido por una expresión condicional. El índice contiene entradas solo para las filas de la tabla que satisfacen el predicado.

La motivación de los índices parciales es evitar indexar valores comunes. Dado que una búsqueda para un valor común no usará el índice de todas maneras no hay necesidad de mantener esas filas en el índice. Esto reduce el tamaño del índice que hará más rápidas aquellas consultas que lo usen así como las actualizaciones de la tabla ya que no será necesario actualizarlo en todos los casos.

- Partial Indexes

Índices multicolumna

Un índice puede ser definido sobre más de una columna de una tabla. Son apropiados cuando hay consultas con predicados por las dos columnas del índice.

- Multicolumn Indexes

Restricciones, *Constraints*

Los tipos de datos son una forma de limitar los tipos de datos que pueden ser almacenados en una tabla. Para muchas aplicaciones las restricciones que proporcionan son demasiado simples. Por ejemplo, una columna que contenga el precio de un producto debería aceptar solo valores positivos. Pero no hay un tipo de datos que acepte solo números positivos. Otro problema es que quizá deseemos restringir el dato de una columna respecto a otras columnas o filas. Por ejemplo, en una tabla que contenga información de un producto el número del producto debería ser único.

SQL permite definir restricciones en columnas y tablas proporcionando el control sobre los datos que deseamos. Si se intentan almacenar datos en una columna que viola una restricción se lanza un error.

```
companies=#

CREATE TABLE company (
ID BIGSERIAL PRIMARY KEY NOT NULL,
NAME VARCHAR(50) UNIQUE NOT NULL,
FOUNDATION DATE NOT NULL,
ADDRESS TEXT,
EMPLOYEES INTEGER CONSTRAINT employees_positive CHECK (EMPLOYEES > 0)
);

INSERT INTO company (name, foundation, address, employees) VALUES
('Dummy', '1970-01-01', 'Unknow', -10);
ERROR: new row for relation "company" violates check constraint "employees_positive"


DETAIL: Failing row contains (1, Dummy, 1970-01-01, Unknow, -10).
```

- Constraints

Tipos de tablas

Si se especifica en la creación de la tabla *TEMPORARY* o *TEMP* esta es creada con una tabla temporal que es eliminada al final de la sesión u opcionalmente al finalizar la transacción actual. Si se especifica *UNLOGGED* es creada como no trazable haciendo que los datos escritos en la tabla no sean escritos en el *write-ahead log* que lo hace considerablemente más rápido que las tablas ordinarias. Sin embargo, no son seguras ante fallos.

- CREATE TABLE
-

PL/pgSQL

PostgreSQL al igual que otras bases de datos ofrece un lenguaje procedural que puede ser usado para crear procedimientos de funciones o *triggers*, añadir estructuras de control al lenguaje SQL, realizar cálculos complejos, hereda todos los tipos de usuario, funciones y operadores, puede ser definido como de confianza por el servidor y es fácil de usar. El lenguaje sql es fácil de aprender y es común a las bases de datos relacionales pero cada sentencia SQL debe ser ejecutada individualmente por el servidor. Esto significa que la aplicación cliente debe enviar cada sentencia al servidor, esperar a que sea procesada, recibir y procesar los resultados, realizar algún cálculo y entonces enviar más sentencias al servidor. Todo esto incurre en comunicación entre procesos y de red si el cliente está en una máquina diferente del servidor de base de datos.

Con PL/pgSQL se puede crear un bloque de computación y una serie de sentencias SQL dentro del servidor de base de datos, tiendo el poder de un lenguaje procedural y la facilidad de SQL pero con un considerable ahorro de comunicación entre cliente y servidor. Las ventajas son evitar viajes entre el servidor y el cliente, resultados inmediatos que no son necesarios convertir y transferir entre el cliente y servidor y múltiples pasos de procesado de las sentencias son evitados. Todo esto resulta en algunos casos un incremento de rendimiento considerable comparado con una aplicación que no usa procedimientos almacenados.

- PL/pgSQL - SQL Procedural Language

Otras

Otros elementos que soporta la base de datos PostgreSQL en el lenguaje SQL son *Grouping Sets*, *ROLLUP*, *CUBE*, Set Returning Functions, tablefunc, búsquedas a texto completo que para casos sencillos no hace falta recurrir a soluciones más especializadas como Elasticsearch, selección y bloqueo de filas con la clásula *FOR UPDATE*, vistas y vistas materializadas entre seguro otras muchas cosas de las que me olvido o desconozco.

Por todas estas características se considera a PostgreSQL una de las bases de datos relacionales más avanzadas existentes. Hay mucha literatura sobre las bases de datos relacionales desde el lenguage SQL en general, libros específicos sobre PostgreSQL o como evitar errores diseño al estructurar la información en tablas y columnas.

En caso de optar por una base de datos NoSQL para persistir la información la base de datos NoSQL MongoDB permite guardar los datos en forma de documentos y conseguir la escalabilidad que las bases de datos relacionales con sus propiedades ACID es difícil.

Referencia:

- Documentación PostgreSQL
- Introducción a la base de datos NoSQL MongoDB
- Usar la base de datos NoSQL MongoDB con Java
- Introducción a la base de datos NoSQL Redis
- Serie de artículos sobre Docker

- Más artículos recientes en el archivo

- Java
- GNU/Linux
- JavaScript
- Tapestry
- Archivo y hemeroteca
- Enlaces
- Publicidad
- Acerca de...