PostgreSQL

Javier Ramos Fernández

Tipos de datos

Primitivos

Numéricos

Name	Storage Size	Description	Range
smallint	2 bytes	small-range integer	-32768 to +32767
integer	4 bytes	typical choice for integer	-2147483648 to +2147483647
bigint	8 bytes	large-range integer	-9223372036854775808 to 9223372036854775807
decimal	variable	user-specified precision,exact	up to 131072 digits before the decimal point; up to 16383 digits after the decimal point
numeric	variable	user-specified precision,exact	up to 131072 digits before the decimal point; up to 16383 digits after the decimal point
real	4 bytes	variable-precision,inexact	6 decimal digits precision
double precision	8 bytes	variable-precision,inexact	15 decimal digits precision
smallserial	2 bytes	small autoincrementing integer	1 to 32767
serial	4 bytes	autoincrementing integer	1 to 2147483647
bigserial	8 bytes	large autoincrementing integer	1 to 9223372036854775807

Monetarios

Name	Storage Size	Description	Range
money	8 bytes	currency amount	-92233720368547758.08 to +92233720368547758.07

Caracteres

Name	Description
character varying(n), varchar(n)	variable-length with limit
character(n), char(n)	fixed-length, blank padded
text	variable unlimited length

Binarios

Name	Storage Size	Description
bytea	1 or 4 bytes plus the actual binary string	variable-length binary string

Fecha/Tiempo

Name	Storage Size	Description	Low Value	High Value
timestamp [(p)] [without time zone]	8 bytes	both date and time (no time zone)	4713 BC	294276 AD
timestamp [(p)] with time zone	8 bytes	both date and time, with time zone	4713 BC	294276 AD
date	4 bytes	date (no time of day)	4713 BC	5874897 AD
time [(p)] [without time zone]	8 bytes	time of day (no date)	00:00:00	24:00:00
time [(p)] with time zone	12 bytes	times of day only, with time zone	00:00:00+1459	24:00:00-1459
interval [fields] [(p)]	12 bytes	time interval	-178000000 years	178000000 years

Booleano

Name	Storage Size	Description
boolean	1 byte	state of true or false

Enumerado

```
CREATE TYPE week AS ENUM ('Mon', 'Tue', 'Wed', 'Thu', 'Fri', 'Sat', 'Sun');
```

Geométricos

Name	Storage Size	Representation	Description
point	16 bytes	Point on a plane	(x,y)
line	32 bytes	Infinite line (not fully implemented)	((x1,y1),(x2,y2))
lseg	32 bytes	Finite line segment	((x1,y1),(x2,y2))
box	32 bytes	Rectangular box	((x1,y1),(x2,y2))
path	16+16n bytes	Closed path (similar to polygon)	((x1,y1),...)
path	16+16n bytes	Open path	[(x1,y1),...]
polygon	40+16n	Polygon (similar to closed path)	((x1,y1),...)
circle	24 bytes	Circle	<(x,y),r> (center point and radius)

Arrays

Declaración

Los arrays se pueden declarar de dos formas distintas:

-

```
CREATE TABLE monthly_savings (
   name text,
   saving_per_quarter integer[],
   scheme text[][]
);
```

- Utilizando la palabra clave “ARRAY”:

```
CREATE TABLE monthly_savings (
   name text,
   saving_per_quarter integer ARRAY[4],
   scheme text[][]
);
```

Inserción

Los valores de un array se pueden insertar como constantes literales, encerrando los valores del elemento entre llaves y separándolos por comas. Un ejemplo es el siguiente:

```
INSERT INTO monthly_savings
VALUES ('Manisha',
'{20000, 14600, 23500, 13250}',
'{{"FD", "MF"}, {"FD", "Property"}}');
```

Acceso

A continuación se muestra un ejemplo para acceder a los arrays. El comando que se da a continuación seleccionará las personas cuyos ahorros sean mayores en el segundo trimestre que en el cuarto trimestre.

```
SELECT name FROM monhly_savings WHERE saving_per_quarter[2] >
saving_per_quarter[4];
```

Modificación

Se puede modificar un array de dos formas distintas:

-

```
UPDATE monthly_savings SET saving_per_quarter = '{25000,25000,27000,27000}'
WHERE name = 'Manisha';
```

-

```
UPDATE monthly_savings SET saving_per_quarter = ARRAY[25000,25000,27000,27000]
WHERE name = 'Manisha';
```

Búsqueda

Un ejemplo de búsqueda de un array es el siguiente:

```
SELECT * FROM monthly_savings WHERE saving_per_quarter[1] = 10000 OR
saving_per_quarter[2] = 10000 OR
saving_per_quarter[3] = 10000 OR
saving_per_quarter[4] = 10000;
```

Si se conoce el tamaño de la matriz, se puede utilizar el método de búsqueda indicado anteriormente. Si no, el siguiente ejemplo muestra cómo buscar un array cuando no se conoce el tamaño.

```
SELECT * FROM monthly_savings WHERE 10000 = ANY (saving_per_quarter);
```

Compuestos

Declaración

El siguiente ejemplo muestra cómo declarar un tipo compuesto:

```
CREATE TYPE inventory_item AS (
    name text,
    supplier_id integer,
    price numeric
);
```

Este tipo de datos se puede utilizar a la hora de crear tablas como se indica a continuación:

```
CREATE TABLE on_hand (
    item inventory_item,
    count integer
);
```

Valores de entrada

Los valores compuestos se pueden insertar como una constante literal, encerrando los valores de campo entre paréntesis y separándolos por comas. A continuación se muestra un ejemplo:

```
INSERT INTO on_hand VALUES (ROW('fuzzy dice', 42, 1.99), 1000);
```

Esto es válido para el *inventory_item* definido anteriormente. La palabra clave **ROW** es en realidad opcional siempre y cuando se tenga más de un campo en la expresión.

Acceso

Para acceder a un campo de una columna compuesta, se utiliza un punto seguido del nombre del campo, al igual que se selecciona un campo de un nombre de tabla. Por ejemplo, para seleccionar algunos subcampos de nuestra tabla de ejemplo *on_hand*, la consulta sería como se muestra a continuación:

```
SELECT (item).name FROM on_hand WHERE (item).price > 9.99;
```

También se puede utilizar el nombre de tabla (por ejemplo, en una consulta multitabla), de la siguiente manera:

```
SELECT (on_hand.item).name FROM on_hand WHERE (on_hand.item).price > 9.99;
```

Rangos

Los tipos de rango representan tipos de datos que utilizan un rango de datos. Los rangos pueden ser rangos **discretos** (por ejemplo, todos los valores enteros 1 a 10) o rangos continuos (por ejemplo, cualquier punto en el tiempo entre las 10:00am y las 11:00am). Los tipos disponibles incluyen los siguientes rangos:

- **int4range** - Rango de enteros
- **int8range** - Rango de *bigint*
- **numrange** - Rango de *numeric*
- **tsrange** - Rango de marcas temporales sin zona horaria
- **tstzrange** - Rango de marcas temporales con zona horaria
- **daterange** - Rango de fechas

Los tipos de rango admiten límites de rango inclusivos y exclusivos utilizando los caracteres **[]** y **()** respectivamente.

Sentencias utilizadas

- Exportar una tabla a un archivo .csv:

  COPY table_to_export TO 'file_path' DELIMITER ',' CSV HEADER;

  Ejemplo:

  COPY road_links_modified FROM '/home/javisunami/Escritorio/TFG/datasetsOriginales/training/links_table3.csv' WITH CSV HEADER;

- Crear una tabla en la base de datos:

  CREATE TABLE table_name(
  column1 datatype,
  column2 datatype,
  column3 datatype,
  .....
  columnN datatype,
  PRIMARY KEY(one or more columns));

Ejemplo:

CREATE TABLE road_links_modified(link_id smallint PRIMARY KEY,
length float,
width float,
lanes int,
in_top varchar(9),
out_top varchar(9),
lane_width float);

- Actualizar los valores de los registros de una tabla:

UPDATE table_name
SET column1 = value1, column2 = value2...., columnN = valueN
WHERE [condition];

Ejemplo:

UPDATE road_links_modified SET out_top = STRING_TO_ARRAY(out_top, ',');

- Alterar una tabla para cambiar el tipo de dato de un atributo de una tabla con conversión explícita:

ALTER TABLE table ALTER column TYPE new_type USING column::new_type;

Ejemplo:

ALTER TABLE road_links_modified ALTER out_top TYPE smallint[] USING out_top::smallint[];

- Eliminar una tabla:

DROP TABLE table_name;

Ejemplo:

DROP TABLE weather_data_modified;

- Definir una restricción *CHECK* en la definición de una columna de una tabla:

  CONSTRAINT constraint_name CHECK (condition)

  Ejemplo:

  ... CONSTRAINT has_tollgate_id_value CHECK (tollgate_id IN (1,2,3)), ...

- Definir una restricción PRIMARY KEY en la creación de una tabla:

  PRIMARY KEY(list_of_columns)

  Ejemplo:

  columns_definitions,
  ...,
  PRIMARY KEY (intersection_id, tollgate_id), ...

- Ejecutar un bloque anónimo:

  DO $$
  <<name_of_the_block>>
  DECLARE
  -- Declaration of variables
  BEGIN
  -- Body of the block
  END <<name_of_the_block>>;
  $$

  Ejemplo :

  DO $$
  <<first_block>>
  DECLARE
  t_row vehicle_trajectories_training_modified%rowtype;
  *curs1 CURSOR FOR SELECT * FROM*
  vehicle_trajectories_training_modified FOR UPDATE;
  link link_object;
  conjunto_links link_object[] DEFAULT '{}';
  campo varchar(100);

```
  array_componentes_campo varchar(40) ARRAY;
BEGIN
  OPEN curs1;
  LOOP
        FETCH curs1 INTO t_row;
        EXIT WHEN t_row IS NULL;
        FOREACH campo IN ARRAY t_row.travel_seq
        LOOP
        array_componentes_campo := STRING_TO_ARRAY(campo, '#');
        link.id :=  array_componentes_campo[1];
        link.entrance_time :=  array_componentes_campo[2];
        link.duration :=  array_componentes_campo[3];
        conjunto_links := conjunto_links || link;
        END LOOP;
        UPDATE vehicle_trajectories_training_modified SET travel_seq =
conjunto_links
                WHERE CURRENT OF curs1;
        conjunto_links := '{}';
        link = null;
  END LOOP;
  CLOSE curs1;
END first_block $$;
```

- Declaración de una variable:

name [CONSTANT] type [COLLATE collation_name] [NOT NULL] [{ DEFAULT | := } expression];

La cláusula DEFAULT, si se da, especifica el valor inicial asignado a la variable cuando el bloque se ejecuta. Si la cláusula DEFAULT no se da, entonces la variable se inicializa al valor nulo SQL. La opción CONSTANT impide que se le vuelva a asignar un valor a la variable, de forma que su valor permanece constante durante la ejecución del bloque. Si se especifica NOT NULL, una asignación de un valor nulo provoca un error en tiempo de ejecución. Todas las variables declaradas NOT NULL deben tener un valor por defecto no nulo especificado.

Ejemplo:

```
conjunto_links link_object[] DEFAULT '{}';
```

- Declaración de un cursor:

name [[NO] SCROLL] CURSOR [(arguments)] FOR query;

Si se especifica *SCROLL*, el cursor será capaz de desplazarse hacia atrás; si se especifica *NO SCROLL* , las búsquedas hacia atrás serán rechazadas; si ninguna de las especificaciones aparece, las búsquedas hacia atrás se permitirán dependiendo de la consulta. *arguments*, si se especifica, es una lista separada por comas de pares *name datatype* que definen nombres para ser reemplazados por valores de parámetro en la consulta dada. Los valores reales a sustituir por estos nombres se especificarán más adelante, cuando se abra el cursor.

Ejemplo:

*curs1 CURSOR FOR SELECT * FROM*
vehicle_trajectories_training_modified FOR UPDATE;

Nota : La opción *FOR UPDATE* se especifica para que, al acceder a las filas de la tabla resultado de la consulta, se puedan actualizar todas las columnas de la misma.

- Apertura de un cursor:

  OPEN cursor_name;

  Ejemplo:

  OPEN curs1;

- Buscar una fila de la tabla que resulta de la realización de la tabla:

  FETCH [direction { FROM | IN }] cursor INTO target;

FETCH recupera la siguiente fila del cursor a un objetivo, que puede ser una variable de fila, una variable de registro o una lista de variables simples separadas por comas, al igual que *SELECT INTO*. Si no hay ninguna fila siguiente, el objetivo se fija a *NULL (s)*. Al igual que con *SELECT INTO*, se puede comprobar la variable especial *FOUND* para ver si se ha obtenido o no una fila.

La cláusula *direction* puede ser cualquiera de las variantes permitidas en el comando *SQL FETCH* excepto las que pueden obtener más de una fila; es decir, puede ser *NEXT*, *PRIOR*, *FIRS*T, *LAST*, *ABSOLUTE count*, *RELATIVE count*, *FORWARD*, o *BACKWAR*D. Omitir *direction* es lo mismo que especificar *NEXT*. Los

valores de d*irection* que requieren retroceder hacia atrás pueden fallar a menos que el cursor se haya declarado o abierto con la opción *SCROLL*.

cursor debe ser el nombre de una variable *refcursor* que hace referencia a un portal de cursor abierto.

Ejemplo:

FETCH curs1 INTO t_row;

- Salir del bucle que realiza las búsquedas de la fila de la tabla resultado de la consulta cuando ya la tabla no tiene más filas:

EXIT WHEN rowtype_variableIS NULL; (Después de haber ejecutado *FETCH cursor INTO variable_tipo_fila_tabla;*)

La declaración de *variable_tipo_fila* se realiza de la siguiente manera:

***variable_tipo_fila tabla*%rowtype**

Ejemplo :

FETCH curs1 INTO t_row;
EXIT WHEN t_row IS NULL;

- Actualizar la fila actual a la que apunta un cursor:

UPDATE table SET column = value WHERE CURRENT OF cursor;

La sentencia *WHERE CURRENT OF* le permite actualizar o eliminar el registro que fue obtenido por última vez por el cursor. Esto es posible si las filas de la tabla resultado de la consuta se han referenciado mediante una sentencia *SELECT FOR UPDATE*.

Ejemplo:

UPDATE vehicle_trajectories_training_modified SET travel_seq = conjunto_links WHERE CURRENT OF curs1;

- Cerrar un cursor:

CLOSE cursor;

Ejemplo:

CLOSE curs1;

- Bucles PostgreSQL utilizados:

  - **<<label>>**
    LOOP
    Statements;
    EXIT [<<label>>] WHEN condition;
    END LOOP;

La sentencia LOOP se conoce como *declaración de bucle incondicional* porque ejecuta las sentencias hasta que la condición en la sentencia *EXIT* evalúa a *true*.

Ejemplo:

LOOP
FETCH curs1 INTO t_row;
EXIT WHEN t_row IS NULL;
...
END LOOP;

La condición en este caso es la que vimos en el apartado anterior.

  - **[<<label>>]**
    FOREACH target [SLICE number] IN ARRAY expression
    LOOP
    statements
    END LOOP [label];

El bucle *FOREACH* es muy parecido a un bucle *FOR*, pero en lugar de iterar a través de las filas devueltas por una consulta *SQL*, itera a través de los elementos de un array.

Sin SLICE, o si se especifica SLICE 0, el loop se itera a través de elementos individuales del array producido evaluando la expresión. A la variable destino se le asigna cada valor de elemento en secuencia, y el cuerpo del bucle se ejecuta para cada elemento.

Ejemplo:

```
FOREACH campo IN ARRAY t_row.travel_seq
LOOP
array_componentes_campo := STRING_TO_ARRAY(campo, '#');
link.id :=  array_componentes_campo[1];
link.entrance_time :=  array_componentes_campo[2];
link.duration :=  array_componentes_campo[3];
conjunto_links := conjunto_links || link;
END LOOP;
```

- Creación de un procedimiento almacenado sin valor de retorno:

```
CREATE [OR REPLACE] FUNCTION function_name (arguments)
RETURNS return_datatype AS $variable_name$
DECLARE
declaration;
[...]
BEGIN
< function_body >
[...]
RETURN { variable_name | value }
END; LANGUAGE plpgsql;
```

- **function-name** especifica el nombre de la función.
- La opción [OR REPLACE] permite modificar una función existente. option allows modifying an existing function.
- La función debe contener una sentencia **return**.
- La cláusula **RETURN** especifica el tipo de dato que se va a retornar desde la función. El tipo **return_datatype** puede ser un tipo de base, compuesto o dominio, o puede hacer referencia al tipo de una columna de una tabla.
- **function-body** contiene la parte ejecutable.
- La palabra clave **AS** se usa para crear una función independiente.
- **plpgsql** es el nombre del lenguaje en el que está implementada la función. Aquí, utilizamos esta opción para PostgreSQL. Puede ser SQL, C, interno, o el nombre de un lenguaje procedural definido por el usuario. Para la compatibilidad hacia atrás, el nombre puede estar encerrado entre comillas simples.

Ejemplo:

```
CREATE OR REPLACE FUNCTION insert_rows(fecha_inicial timestamp, fecha_final
timestamp)
        RETURNS void AS $$
        DECLARE
        tollgates integer[3] DEFAULT '{1,2,3}';
        directions integer[2] DEFAULT '{0,1}';
        tollgate_id integer;
        direction integer;
        aux timestamp;
        contador integer;
        BEGIN
        FOREACH tollgate_id IN ARRAY tollgates
        LOOP
        FOREACH direction IN ARRAY directions
        LOOP
        aux := fecha_inicial;
        WHILE aux != fecha_final LOOP
        FOR contador in 1..7 LOOP
                INSERT INTO  tabla_resultado_traffic_volume VALUES(tollgate_id,
ARRAY[aux, aux + '20 minute'],direction,NULL);
                aux :=  aux + '1 day';
        END LOOP;
        aux := aux + '-7 days';
        aux := aux + '20 minute';
        END LOOP;
        END LOOP;
        END LOOP;
        END;
        $$ LANGUAGE plpgsql;
```

En este caso la función no retorna nada, por lo que se trata de un **procedimiento**.

- Creación de una vista:

CREATE [OR REPLACE] [TEMP | TEMPORARY] VIEW name [(column_name [, ...])]
[WITH (view_option_name [= view_option_value] [, ...])]
AS query

- **CREATE VIEW** define una vista de una consulta. La vista no se materializa físicamente; en su lugar, la consulta se ejecuta cada vez que se hace referencia a la vista en una consulta.

- **CREATE OR REPLACE VIEW** es similar, pero si ya existe una vista del mismo nombre, se reemplaza. La nueva consulta debe generar las mismas columnas generadas por la consulta de vista existente (es decir, los mismos nombres de columna en el mismo orden y con los mismos tipos de datos), pero puede añadir columnas adicionales al final de la lista. Los cálculos que dan lugar a las columnas de salida pueden ser completamente diferentes.

- Si se da un nombre de esquema (por ejemplo, **CREATE VIEW myschema. myview** ...), la vista se crea en el esquema especificado. De lo contrario, se crea en el esquema actual. Las **vistas temporales** existen en un esquema especial, por lo que no se puede dar un nombre de esquema al crear una vista temporal.

- El nombre de la vista debe ser distinto del nombre de cualquier otra vista, tabla, secuencia, índice o tabla extranjera en el mismo esquema.

  Ejemplo:

```
CREATE VIEW comedies AS
 SELECT *
 FROM films
 WHERE kind = 'Comedy';
```

❖ Extraer la hora de una determinada fecha:

EXTRACT(field FROM source)

La función *extract* extrae subcampos como, por ejemplo, **valores de fecha/hora del año** u **hora**. *source* debe ser una expresión de valor del tipo **timestamp**, **time** o **interval**. (Expresiones de tipo **date** serán transformadas a **timestamp** y por lo tanto pueden ser usadas también.) *field* es un identificador o cadena que selecciona qué campo extraer del valor fuente. La función *extract* devuelve valores de doble precisión.

Ejemplo:

El campo *hora* (0 - 23)

SELECT EXTRACT(HOUR FROM TIMESTAMP '2001-02-16 20:38:40');
Result: 20

- Combinación interna de dos tablas:

SELECT fields
FROM table1
JOIN table2
ON combination_condition;

Ejemplo:
SELECT * FROM libros
JOIN editoriales
ON codigoeditorial=editoriales.codigo;

La consulta anterior se interpreta de la siguiente manera.

- Especificamos los campos que aparecerán en el resultado en la lista de selección.
- Indicamos el nombre de la tabla después del **FROM** (*libros*).
- Combinamos esa tabla con **JOIN** y el nombre de la otra tabla (*editoriales*); se especifica qué tablas se van a combinar y cómo.
- Cuando se combina información de varias tablas, es necesario especificar qué registro de una tabla se combinará con qué registro de la otra tabla con la cláusula **ON**. Se debe especificar la condición para enlazarlas, es decir, el campo por el cual se combinarán, que tienen en común. La cláusula **ON** hace coincidir registros de ambas tablas basándose en el valor de tal campo. En el ejemplo, el campo *codigoeditorial* de *libros* y el campo *codigo* de *editoriales* son los que enlazarán ambas tablas. Se emplean campos comunes, que deben tener tipos de datos iguales o similares.

La condición de combinación, es decir, el o los campos por los que se van a combinar (cláusula **ON**), se especifica según las claves primarias y externas. Note que en la consulta, al nombrar el campo usamos el nombre de la tabla también. Cuando las tablas referenciadas tienen campos con igual nombre, esto es necesario para evitar confusiones y ambigüedades al momento de referenciar un campo. En el ejemplo, si no especificamos *editoriales.codigo* y solamente escribimos *codigo*, PostgreSQL no sabrá si nos referimos al campo

codigo de *libros* o de *editoriales* y mostrará un mensaje de error indicando que *codigo* es ambiguo.

❖ Eliminar una vista:

DROP VIEW name [, ...] [CASCADE | RESTRICT]

- *name*: El nombre (opcionalmente calificado por el esquema) de la vista a eliminar.
- *CASCADE*: Eliminar automáticamente objetos que dependen de la vista (por ejemplo, otras vistas).
- *RESTRICT*: Negarse a dejar eliminar la vista si algún objeto depende de ella. Este es el valor predeterminado.

Ejemplo:

DROP VIEW kinds;

Este ejemplo elimina la vista *kinds*.

❖ Eliminar todas las vistas de una base de datos:

SELECT 'DROP VIEW ' || table_name || ';'
FROM information_schema.views
WHERE table_schema NOT IN ('pg_catalog', 'information_schema')
AND table_name !~ '^pg_';

Meta-comandos utilizados

- Limitar la longitud de lineas largas en las tablas:

  \pset format wrapped

- Salir del programa *psql*:

  \q

- Obtener un listado de las tablas que forman la base de datos:

  \d o **\dt**

- Obtener el esquema de una tabla:

  \d table

- Leer la entrada del nombre de archivo y ejecutarlo como si se hubiera escrito en el teclado:

  \i filename

Acceso remoto a PostgreSQL

En el ordenador servidor:

1. Acceder al fichero **/etc/postgresql/9.3/main/pg_hba.conf** con permisos de root.
2. Añadir una línea que sea del tipo **host all all [ip_ordenador_cliente]/32 md5**.
3. Ejecutar la sentencia **sudo service postgresql restart**.

En el ordenador cliente, ejecutar la sentencia **psql -h [ip_servidor] -U [nombre_usuario] -d [base_de_datos]**.

Bibliografía

- https://wiki.postgresql.org/wiki/Psycopg2_Tutorial
- https://www.postgresql.org/docs/9.3/static/