PREVIRED

05/2016

Funciones

Funciones

```
CREATE [ OR REPLACE ] FUNCTION nombre_funcion([ [ argmodo ] [ argnombre ]
argtipo [, ...] ]) RETURNS tipo AS $$
[ DECLARE ] [ declaraciones de variables ]
BEGIN
codigo
END;
$$ LANGUAGE plpgsql | IMMUTABLE | STABLE | VOLATILE | CALLED ON NULL
INPUT | RETURNS NULL ON NULL INPUT | STRICT | [ EXTERNAL ] SECURITY INVOKER
| [ EXTERNAL ] SECURITY DEFINER | COST execution_cost | ROWS result_rows
| SET configuration_parameter { TO value | = value | FROM CURRENT };
```

Funciones

- argmodo: El modo de un argumento puede ser IN, OUT, or INOUT. Por defecto se usa IN si no se define.
- argtipo: Los tipos que podemos utilizar son todos los disponibles en PostgreSQL y todos los definidos por el usuario
- Las declaraciones de variables se pueden realizar de la siguiente manera ($n = orden de declaración del argumento.):

```
nombre_variable ALIAS FOR $n;
nombre_variable [ CONSTANT ] tipo [ NOT NULL ] [ { DEFAULT | :=
} expresion ];
```

Funciones

LANGUAGE

- –El lenguaje debe ser uno instalado en la base de datos.
- –Para obtener un listado de los lenguajes
  `SELECT lanname FROM pg_language.`

•Volatilidad.Este parámetro le da un "tip" al query planner para que determine que las salidas pueden ser almacenadas en el caché y reutilizadas para múltiples llamadas.

IMMUTABLE

- –La función siempre retorna la misma salida para la misma entrada

STABLE

- –Indica que la función no puede alterar a la base de datos y que siempre devolverá el mismo resultado en una consulta individual de una tabla, dados los mismos valores como argumentos. El resultado podria cambiar entre sentencias SQL.

VOLATILE

- –La función retorna distintos valores , aún cuando la entrada sea igual.
- –Este es el default
- –Funciones que cambian datos o que dependen de variables ambientales como la fecha del sistema deben ser marcadas como volátiles.
- –Considerar que el query planner de todas formas puede considerar ejecutar la función si eso es más eficiente en términos de costo.

Funciones

`CALLED ON NULL INPUT | RETURNS NULL ON NULL INPUT | STRICT`
`CALLED ON NULL INPUT`
Indica que la función se ejecutará aunque algunos de los argumentos sean NULL. El usuario tiene la responsabilidad de comprobar si algún argumento es NULL cuando sea necesario tener esto en cuenta.(valor por defecto)
`RETURNS NULL ON NULL INPUT / STRICT`
Indican que la función no se ejecutará y devolverá el valor NULL si alguno de los argumentos es NULL.

Funciones

COST

- –Es una medida relativa de la intensidad del cómputo. SQL y PL/pgSQL
- –Tienen un valor de 100 y las funciones en 1. Esto afecta el ordenque sigue el planner cuando evalua las funciones en una claúsula WHERE clause y por lo tanto la necesidad de hacer caché del resultado.Mientras mayor el valor mayor es el costo que asume el planner

ROWS

- –Aplica solo a funciones que retornan un conjunto de registros. Este valor provee una estimacón de cuantas filas se retornarán.Esto permite al planner tomar la mejor estrategia para invocar la función
- –SECURITY DEFINER
- –Esto ejecuta la función en el contexto del owner. Si se omite se ejecuta en el contexto de seguridad del usuario que la invoca.
- –Esto es útil para actualizar tablas donde al usuario no se le otorgan privilegios

Funciones

```
CREATE OR REPLACE FUNCTION ejemplo() RETURNS integer AS $$
BEGIN
      RETURN 104;
END;
$$ LANGUAGE plpgsql;

SELECT ejemplo();
ejemplo1 ----------
104(1 row)

CREATE OR REPLACE FUNCTION ejemplo(integer) RETURNS integer AS $$
BEGIN
      RETURN $1;
END;
$$ LANGUAGE plpgsql;
```

Funciones

```
CREATE OR REPLACE FUNCTION ejemplo(numero integer) RETURNS integer AS $$
BEGIN
RETURN numero;
END;
$$ LANGUAGE plpgsql;

CREATE OR REPLACE FUNCTION ejemplo(integer) RETURNS integer AS $$
DECLARE numero ALIAS FOR $1;
BEGIN
RETURN numero;END;
$$ LANGUAGE plpgsql;
```

Funciones

```
CREATE OR REPLACE FUNCTION ejemplo(integer, integer) RETURNS integer AS
$$
DECLARE
      numero1 ALIAS FOR $1;
      numero2 ALIAS FOR $2;
      constante CONSTANT integer := 100;
      resultado integer;
BEGIN
      resultado := (numero1 * numero2) + constante;
      RETURN resultado;
END;
$$ LANGUAGE plpgsql;
```

Funciones

```
CREATE OR REPLACE FUNCTION ejemplo_txt(integer, integer) RETURNS text AS $$
DECLARE
      numero1 ALIAS FOR $1;
      numero2 ALIAS FOR $2;
      constante CONSTANT integer := 100;
      resultado INTEGER;
      resultado_txt TEXT DEFAULT 'El resultado es 104';
BEGIN
      resultado := (numero1 * numero2) + constante;
      IF resultado <> 104 THEN
               resultado_txt := 'El resultado NO es 104';
      END IF;
      RETURN resultado_txt;
END;
$$ LANGUAGE plpgsql;
```

Funciones

```
CREATE OR REPLACE FUNCTION fn_sqltestout(param_subject text,     OUT
subject_scramble text, OUT subject_char text)   AS
$$
SELECT  substring($1, 1,CAST(random()*length($1) As integer)),
substring($1, 1,1)
$$  LANGUAGE 'sql' VOLATILE;
SELECT  (fn_sqltestout('This is a test subject')).subject_scramble;
SELECT  (fn_sqltestout('This is a test subject')).*;
```

Funciones

```
CREATE OR REPLACE FUNCTION fn_plpgsqltestout(param_subject text, OUT
subject_scramble text, OUT subject_char text)   AS
$$
BEGIN
      subject_scramble := substring($1, 1,CAST(random()*length($1) As
integer));
      subject_char := substring($1, 1,1);
END;
 $$  LANGUAGE 'plpgsql' VOLATILE;
```

Funciones

```
CREATE OR REPLACE FUNCTION fn_sqltestmulti(param_subject varchar,
OUT test_id integer,     OUT test_stuff text)     RETURNS SETOF record
AS
$$
      SELECT test_id, test_stuffFROM testtable where test_stuff LIKE $1;
$$
LANGUAGE 'sql' VOLATILE;

SELECT * FROM fn_sqltestmulti('%stuff%');
```

Funciones

```
CREATE OR REPLACE FUNCTION fn_plpgsqltestmulti(     param_subject
varchar,      OUT test_id integer,      OUT test_stuff text)      RETURNS
SETOF record    AS
$$
BEGIN
      RETURN QUERY SELECT t.test_id , t.test_stuff  FROM testtable As t
WHERE t.test_stuff  LIKE param_subject;
END;
$$  LANGUAGE 'plpgsql'  VOLATILE;
```

Funciones

- CREATE OR REPLACE FUNCTION fn_sqltestmulti(param_subject varchar, OUT test_id integer, OUT test_stuff text) RETURNS SETOF record AS$$ SELECT test_id, test_stuff FROM testtable where test_stuff LIKE $1;$$ LANGUAGE 'sql' VOLATILE;

Funciones

```
CREATE TABLE test001 (id integer,value  text);

INSERT INTO test001 VALUES (1,'a');
INSERT INTO test001 VALUES (1,'b');
INSERT INTO test001 VALUES (1,'c');

SELECT * from test001 ;

CREATE OR REPLACE FUNCTION show_data() RETURNS SETOF test001 AS $$
DECLARE
    sql_result test001%rowtype;
BEGIN
      FOR sql_result in EXECUTE 'SELECT * from test001' LOOP
               RETURN NEXT sql_result;
      END LOOP;
END;
$$ LANGUAGE plpgsql;
```

Triggers

- Los triggers son fundamentales para detectar cambios en los datos y en PostgreSQL se pueden atachar a tablas y vistas.
- Los Triggers actúan a nivel de instrucciones y a nivel de filas.
- A nivel de instrucciones los triggers se ejecuta una vez por instruccción SQL.
- A nivel de filas se ejecuta por cada fila run for each row affected by the SQL.
- Por ejemplo si se ejecuta una instrucción UPDATE que afecta a 1.500 filas, a nivel de instrucción el trigger es ejecutado una vez, a nivel de filas 1500 veces
- Obviamente es factible controlar cuando es ejecutado el trigger BEFORE, AFTER, y INSTEAD OF.
- BEFORE ejecuta el trigger antes de ejecutar la instrucción por lo que se puede cancelar o respaldar datos.
- AFTER triggers se utilizan para respaldo o replicación
- INSTEAD OF ejecutan el trigger en vez del la instrucción
- BEFORE y AFTER solo se utilizan con tablas
- INSTEAD OF solo con vistas.

Triggers

- Al trigger se le puede agregar una condición WHEN, para controlar cuando es gatillado.
- También se puede usar UPDATE OF, para que solo sea gatillado cuando ciertas columnas son actualizadas.
- PostgreSQL ofrcen las funciones especializadas para trigger que se comportan como cualquier otra función pero no tienen argumentos de entrada y el tipo de salida es fijo (De tipo trigger)
- Una función de tipo trigger puede ser reutilizada en disntintos triggers.
- En PostgreSQL, cada trigger debe tener asociada una función de trigger para ser gatillada.
- Para tener múltiples funciones se deben crear múltiples triggers asociados al mismo evento.
- El orden alfábetivo de acuerdo al nombre del trigger, es el orden de ejecución.
- Cada trigger tiene acceso a los datos modificados por el trigger anterior.
- Los triggers no son ejecutados en transacciones seperadas, por lo que si cualquier trigger hace un rollback, se hará un rollback de todos los triggers disparados en el evento.
- Se puede utilizar cualquier lenguaje para crear un trigger, pero habitualmente se usa PG/SQL

Triggers

- Ejemplo tabla con apellido_paterno, apellido_materno y nombres. Se quiere actualizar el campo nombre.

```
CREATE OR REPLACE FUNCTION actualizar_nombre()  RETURNS TRIGGER AS $trigger_ejemplo$
BEGIN
NEW.nombre := NEW.apellido_paterno || ' ' || NEW.apellido_materno || ' ' || NEW.nombres ;
RETURN NEW;
END;
$trigger_ejemplo$ LANGUAGE plpgsql;

CREATE TRIGGER trigger_ejemplo BEFORE INSERT OR UPDATE ON tabla FOR EACH ROW EXECUTE
PROCEDURE actualizar_nombre();
```

Triggers

- NEW Tipo de dato RECORD; Variable que contiene la nueva fila de la tabla para las operaciones INSERT/UPDATE en disparadores del tipo row-level. Esta variable es NULL en disparadores del tipo statement-level.
- OLD Tipo de dato RECORD; Variable que contiene la antigua fila de la tabla para las operaciones UPDATE/DELETE en disparadores del tipo row-level. Esta variable es NULL en disparadores del tipo statement-level.
- TG_NAME Tipo de dato name; variable que contiene el nombre del disparador que está usando la función actualmente.
- TG_WHEN Tipo de dato text; una cadena de texto con el valor BEFORE o AFTER dependiendo de como el disparador que está usando la función actualmente ha sido definido
- TG_LEVEL Tipo de dato text; una cadena de texto con el valor ROW o STATEMENT dependiendo de como el disparador que está usando la función actualmente ha sido definido
- TG_OP Tipo de dato text; una cadena de texto con el valor INSERT, UPDATE o DELETE dependiendo de la operación que ha activado el disparador que está usando la función actualmente.

Triggers

- TG_RELID Tipo de dato oid; el identificador de objeto de la tabla que ha activado el disparador que está usando la función actualment
- TG_RELNAME Tipo de dato name; el nombre de la tabla que ha activado el disparador que está usando la función actualmente. Esta variable es obsoleta y puede desaparacer en el futuro. Se debe usar TG_TABLE_NAME.
- TG_TABLE_NAME Tipo de dato name; el nombre de la tabla que ha activado el disparador que está usando la función actualmente.
- TG_TABLE_SCHEMA Tipo de dato name; el nombre de la schema de la tabla que ha activado el disparador que está usando la función actualmente

Triggers

- Impedir borrar datos

```
CREATE TABLE numeros(  numero bigint NOT NULL,  cuadrado bigint,  cubo bigint,  raiz2
real,  raiz3 real,  PRIMARY KEY (numero));

CREATE OR REPLACE FUNCTION proteger_datos() RETURNS TRIGGER AS $proteger_datos$
DECLARE
BEGIN
-- Esta funcion es usada para proteger datos en un tabla
-- No se permitira el borrado de filas si la usamos
-- en un disparador de tipo BEFORE / row-level
--
RETURN NULL;
END; $proteger_datos$ LANGUAGE plpgsql;

CREATE TRIGGER proteger_datos BEFORE DELETE     ON numeros FOR EACH ROW     EXECUTE
PROCEDURE proteger_datos();
```

Trigger

```
CREATE OR REPLACE FUNCTION rellenar_datos() RETURNS TRIGGER AS
$rellenar_datos$
DECLARE
BEGIN
NEW.cuadrado := power(NEW.numero,2);
NEW.cubo := power(NEW.numero,3);
NEW.raiz2 := sqrt(NEW.numero);
NEW.raiz3 := cbrt(NEW.numero);
RETURN NEW;
 END;
$rellenar_datos$ LANGUAGE plpgsql;

CREATE TRIGGER rellenar_datos BEFORE INSERT OR UPDATE ON numeros FOR
EACH ROW EXECUTE PROCEDURE rellenar_datos();
```

Triggers

```
CREATE OR REPLACE FUNCTION proteger_y_rellenar_datos()  RETURNS TRIGGER AS
$proteger_y_rellenar_datos$
DECLARE
BEGIN
IF (TG_OP = 'INSERT' OR TG_OP = 'UPDATE' ) THEN
      NEW.cuadrado  := power(NEW.numero,2);
      NEW.cubo  := power(NEW.numero,3);
      NEW.raiz2  := sqrt(NEW.numero);
      NEW.raiz3  := cbrt(NEW.numero);
      RETURN NEW;
ELSEIF (TG_OP = 'DELETE') THEN
      RETURN NULL;
END IF;
END;
$proteger_y_rellenar_datos$ LANGUAGE plpgsql;

CREATE TRIGGER proteger_y_rellenar_datos BEFORE INSERT OR UPDATE OR DELETE
ON numeros FOR EACH ROW     EXECUTE PROCEDURE proteger_y_rellenar_datos();
```

Triggers

```
--Ahora  a nivel de instrucciones
CREATE TABLE cambios(
timestamp_ TIMESTAMP WITH TIME ZONE default NOW(),
nombre_disparador text,
tipo_disparador text,
nivel_disparador text,
comando text);
```

Triggers

```
CREATE OR REPLACE FUNCTION grabar_operaciones() RETURNS TRIGGER AS
$grabar_operaciones$
DECLARE
BEGIN
      INSERT INTO cambios (nombre_disparador, tipo_disparador,
                        nivel_disparador,comando)
      VALUES (TG_NAME, TG_WHEN, TG_LEVEL, TG_OP);
RETURN NULL;
END;$grabar_operaciones$ LANGUAGE plpgsql;

CREATE TRIGGER grabar_operaciones AFTER INSERT OR UPDATE OR DELETE    ON
numeros FOR EACH STATEMENT     EXECUTE PROCEDURE grabar_operaciones();
```

Aggregates

- Las funciones más usadas de aggregate del estándar ANSI SQL son MIN,
- MAX, AVG, SUM, y COUNT.
- En PostgreSQL es factible crear nuevos aggregates
- Estas funciones se crean como una window function.
- Los aggregates se pueden escribir en cualquier lenguaje incluyendo SQL por supuesto.
- Un aggregate está compuesto por una o más funciones.
- Debe tener al menos una función que maneje la transición entre estados, y esta es ejecutada fila, por fila, pero es factible tener fucnioens especiales para manejar el estado inicial y el final.

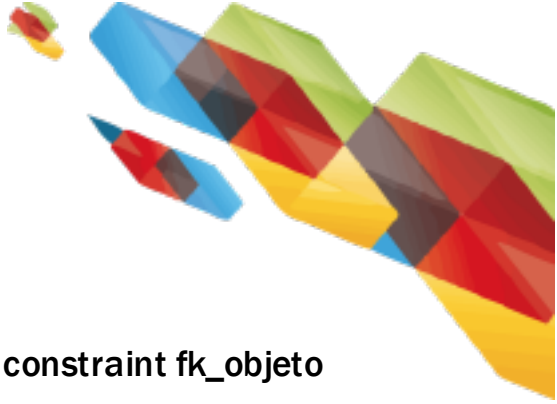
Aggregates

- El comando para crear un aggregate es CREATE AGGREGATE

```
CREATE AGGREGATE my_agg (tipo de dato de entrada) (
SFUNC=nombre de la funcion de estado,
STYPE=tipo de estado,
FINALFUNC=funcion final,
INITCOND=valor del estado inicial, SORTOP=operador
);
```

- La función final debe tomar como entrada el resultado de la funcion de estado.
- La función de estado toma como entrada un dato del tipo de entrada y el resultado de la última llamada a la función de estado.
- La condición inicial es opcional y se inicializa el valor de estado inicial
- El operador asociado de orden sirve para ordenar los datos y tomar ventaja de los indices. (Más adelante veremos esto)

Aggregates



- CREATE TABLE OBJETO (id serial primary key, nombre unique text);
- CREATE TABLE ETIQUETA (id primary key, id_objeto integer, etiqueta text, constraint fk_objeto foreign key (id_objeto) references objeto(id));

Aggregates

```
CREATE OR REPLACE FUNCTION concat(text, text)RETURNS text AS $$
DECLARE
t text;
BEGIN
      IF character_length($1) > 0 THEN
               IF character_length($2) > 0 THEN
                        t = $1 ||', '|| $2;
               ELSE
                        t = $1;
               END IF;
      ELSE
               t = $2;
      END IF;
RETURN t;
END;
$$
LANGUAGE 'plpgsql' VOLATILE;
```

Aggregates

CREATE AGGREGATE concatena(
BASETYPE=text,
SFUNC=concat,
STYPE=text);

```
SELECT nombre,etiqueta as etiquetas FROM objeto JOIN etiquetas USING
(id_objeto) GROUP BY nombre,etiqueta ORDER BY nombre;

SELECT nombre,concatena(etiqueta) AS etiquetas FROM objeto JOIN
etiquetas USING (id_objeto) GROUP BY nombre ;
```

Aggregates

```
CREATE AGGREGATE array_accum(
BASETYPE=anyelement,
SFUNC=array_append,
STYPE=anyarray,
INITCOND='{}');

SELECT nombre,array_accum(etiqueta) AS etiquetas FROM objeto JOIN
etiquetas USING (id_objeto) GROUP BY nombre ;
```

Funciones en SQL

```
CREATE OR REPLACE FUNCTION write_to_log(param_user_name varchar,
param_description
text)
RETURNS integer AS
$$
INSERT INTO logs(user_name, description) VALUES($1, $2)
RETURNING log_id;
$$
LANGUAGE 'sql' VOLATILE;

--llamada
SELECT write_to_log('alejandro', 'Woke up at noon.') As new_id;
```

Funciones en SQL

```
CREATE OR REPLACE FUNCTION
update_logs(log_id int, param_user_name varchar, param_description text)
RETURNS void AS
$$
UPDATE logs SET user_name = $2, description = $3
, log_ts = CURRENT_TIMESTAMP WHERE log_id = $1;
$$
LANGUAGE 'sql' VOLATILE;

--llamada
SELECT update_logs(12, 'alejandro', 'Se quedo dormido')
```

Funciones en SQL

```
CREATE OR REPLACE FUNCTION select_logs_rt(param_user_name varchar)
RETURNS TABLE (log_id int, user_name varchar(50), description text,
log_ts time
stamptz) AS
$$
SELECT log_id, user_name, description, log_ts FROM logs WHERE user_name
= $1;
$$
LANGUAGE 'sql' STABLE;

--parametros de salida
CREATE OR REPLACE FUNCTION select_logs_out(param_user_name varchar, OUT
log_id int
, OUT user_name varchar, OUT description text, OUT log_ts timestamptz)
RETURNS SETOF record AS
$$
SELECT * FROM logs WHERE user_name = $1;
$$
LANGUAGE 'sql' STABLE;
```

Funciones en SQL

- **Uso de un tipo compuesto**

```
CREATE OR REPLACE FUNCTION select_logs_so(param_user_name varchar)
RETURNS SETOF logs AS
$$
SELECT * FROM logs WHERE user_name = $1;
$$
LANGUAGE 'sql' STABLE;
Call all these functions using:
SELECT * FROM select_logs_xxx('alejandro');
```

Funciones y Aggregates

```
CREATE OR REPLACE FUNCTION geom_mean_state(prev numeric[2], next
numeric)
RETURNS numeric[2] AS
$$
SELECT
CASE
WHEN $2 IS NULL OR $2 = 0 THEN $1
ELSE ARRAY[COALESCE($1[1],0) + ln($2), $1[2] + 1]
END;
$$
LANGUAGE sql IMMUTABLE;
```

Funciones y Aggregates

```
CREATE OR REPLACE FUNCTION geom_mean_final(numeric[2])
RETURNS numeric AS
$$
SELECT CASE WHEN $1[2] > 0 THEN exp($1[1]/$1[2]) ELSE 0 END;
$$
LANGUAGE sql IMMUTABLE;
```

Funciones y Aggregate

```
CREATE AGGREGATE geom_mean(numeric) (
SFUNC=geom_mean_state,
STYPE=numeric[],
FINALFUNC=geom_mean_final,
INITCOND='{0,0}'
);

SELECT left(tract_id,5) As county, geom_mean(val) As div_county
FROM census.vw_facts
WHERE category = 'Population' AND short_name != 'white_alone'
GROUP BY county
ORDER BY div_county DESC LIMIT 5;
```

Funciones PL/pgSQL

- Cuando se requiere algo más que SQL, utilizar PL/pgSQL es lo más común.
- PL/pgSQL extiende SQL permitiendo el uso de variables locales vía DECLARE y además el control de flujo.

```
CREATE FUNCTION select_logs_rt(param_user_name varchar)
RETURNS TABLE (log_id int, user_name varchar(50), description text,
log_ts time
stamptz) AS
$$
BEGIN RETURN QUERY
SELECT log_id, user_name, description, log_ts FROM logs
WHERE user_name = param_user_name;
END;
$$
LANGUAGE 'plpgsql' STABLE;
```

Funciones PL/pgSQL

```
CREATE OR REPLACE FUNCTION trig_time_stamper() RETURNS trigger AS
$$
BEGIN
NEW.upd_ts := CURRENT_TIMESTAMP;
RETURN NEW;
END;
$$
LANGUAGE plpgsql VOLATILE;
CREATE TRIGGER trig_1
BEFORE INSERT OR UPDATE OF session_state, session_id
ON web_sessions
FOR EACH ROW EXECUTE PROCEDURE trig_time_stamper();
```

Otras funciones

```
CREATE EXTENSION plpython2u;
CREATE OR REPLACE FUNCTION postgresql_help_search(param_search text)
RETURNS text AS
$$
import urllib, re
response = urllib.urlopen(
'http://www.postgresql.org/search/?u=%2Fdocs%2Fcurrent%2F&q=' +
param_search
)
raw_html = response.read()
result = raw_html[raw_html.find("<!-- docbot goes here -->") :
raw_html.find("<!--pgContentWrap -->") - 1]
result = re.sub('<[^<]+?>', '', result).strip()
return result
$$
LANGUAGE plpython2u SECURITY DEFINER STABLE;
```

Otras funciones

```
CREATE EXTENSION plv8;
CREATE FUNCTION plv8_test(keys text[], vals text[]) RETURNS text
AS $$
var o = {};
for(var i=0; i<keys.length; i++){
   o[keys[i]] = vals[i];
}
return JSON.stringify(o);
$$ LANGUAGE plv8 IMMUTABLE STRICT;

SELECT plv8_test(ARRAY['name', 'age'], ARRAY['Tom', '29']);
plv8_test
---------------------------
{"name":"Tom","age":"29"}
```

Otras funciones

```
CREATE OR REPLACE FUNCTION
validate_email(email text) returns boolean as
$$
var re = /\S+@\S+\.\S+/;
return re.test(email);
$$ LANGUAGE plv8 IMMUTABLE STRICT;
--Codigo Javascript
--Invocacion
SELECT email, validate_email(email) AS is_valid
FROM (VALUES ('alexgomezq@gmail.com')
,('alexgomezqgmail.com'),('alexgomezq@gmailcom')) AS x (email);

email | is_valid
---------------------+----------
alexgomezq@gmail.com | t
alexgomezqgmail.com | f
alexgomezq@gmailcom | f
```