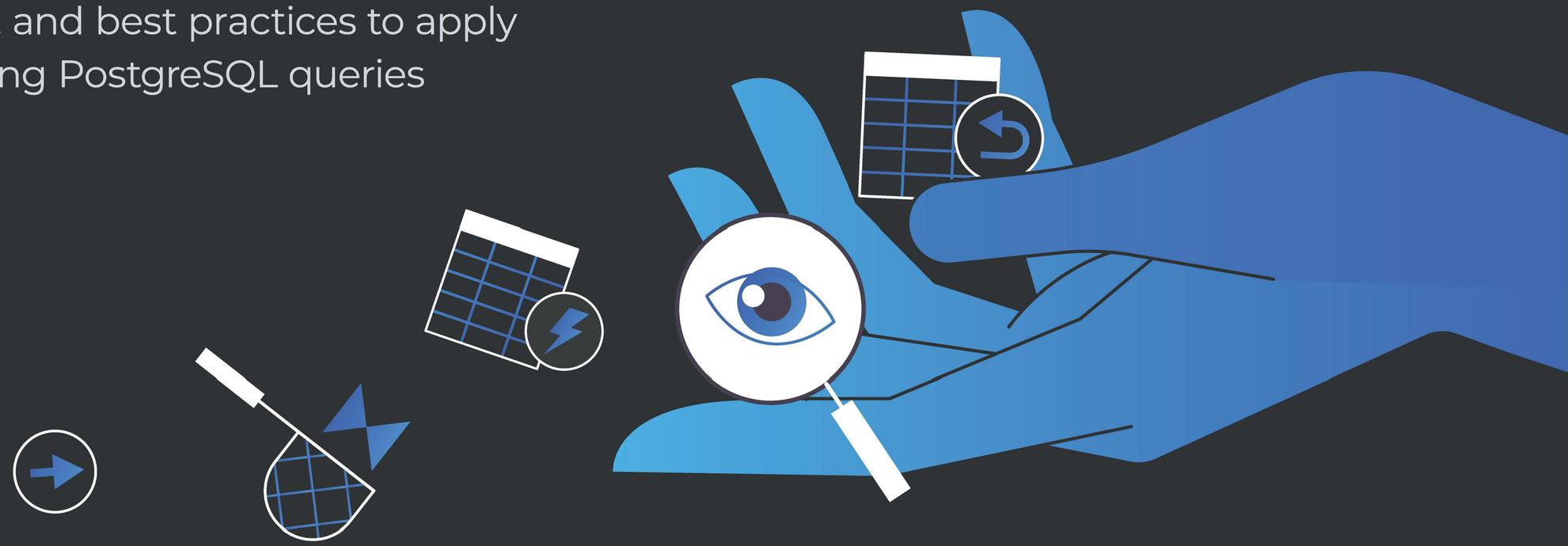
devart

Best practices for writing PostgreSQL queries

Tips, tricks, and best practices to apply when writing PostgreSQL queries



devart

Contents


SQL REPLACE 01


SQL UPDATE with JOIN 03


SQL INSERT INTO SELECT 05

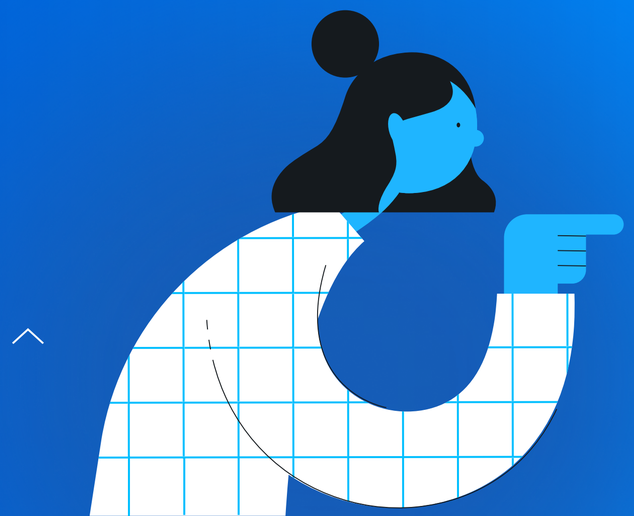

SQL DELETE 07


PostgreSQL Index Examples 08

devart

SQL REPLACE

5 tips for finding and replacing text in PostgreSQL



1. Case-sensitive REPLACE

```
-- Use the default case-sensitive REPLACE:

DO $$
DECLARE
  txt TEXT:= 'Cats are great pets and so easy to take care of.
              They make good companions. Having a cat around
              is good for children.';
  txt2 TEXT;
 BEGIN
   txt2:= REPLACE(txt,'cat','dog');
   RAISE NOTICE '%', txt2;
 END;
$$;

-- Output: Cats are great pets and so easy to take care of. They make good
companions. Having a dog around is good for children.



-- Use the LOWER() function to convert a string to lowercase:

DO $$
DECLARE
  txt TEXT := LOWER('Cats are great pets and so easy to take care of.
                     They make good companions. Having  a cat around
                       is good for children.');
  txt2 TEXT;
 BEGIN
  txt2:= REPLACE(txt,'cat','dog');
  RAISE NOTICE '%', txt2;
 END;
$$;

-- Output: dogs are great pets and so easy to take care of. They make good
companions. Having a dog around is good for children.
```

2. SQL REPLACE can be nested

```
DO $$
DECLARE
  txt TEXT:= 'Cats are great pets and so easy to take care of.
              They make good companions. Having a cat around
                is good for children.';
  txt2 TEXT;
 BEGIN
  txt2:= REPLACE(REPLACE(txt,'cat','dog'),'Cat','Dog');
  RAISE NOTICE '%', txt2;
 END;
$$;

-- Output: Dogs are great pets and so easy to take care of. They
make good companions. Having a dog around is good for children.
```

3. SQL REPLACE replaces text for ALL occurrences

```
-- Replace all occurrences of 'know' within a given string
SELECT REPLACE('know the unknown','know','foresee');
-- Output: 'foresee the unforeseen'



-- Add a space to the string to search for entire words only
SELECT REPLACE('know the unknown','know ','foresee');
-- Output: 'foresee the unknown'
```

4. SQL REPLACE can remove texts

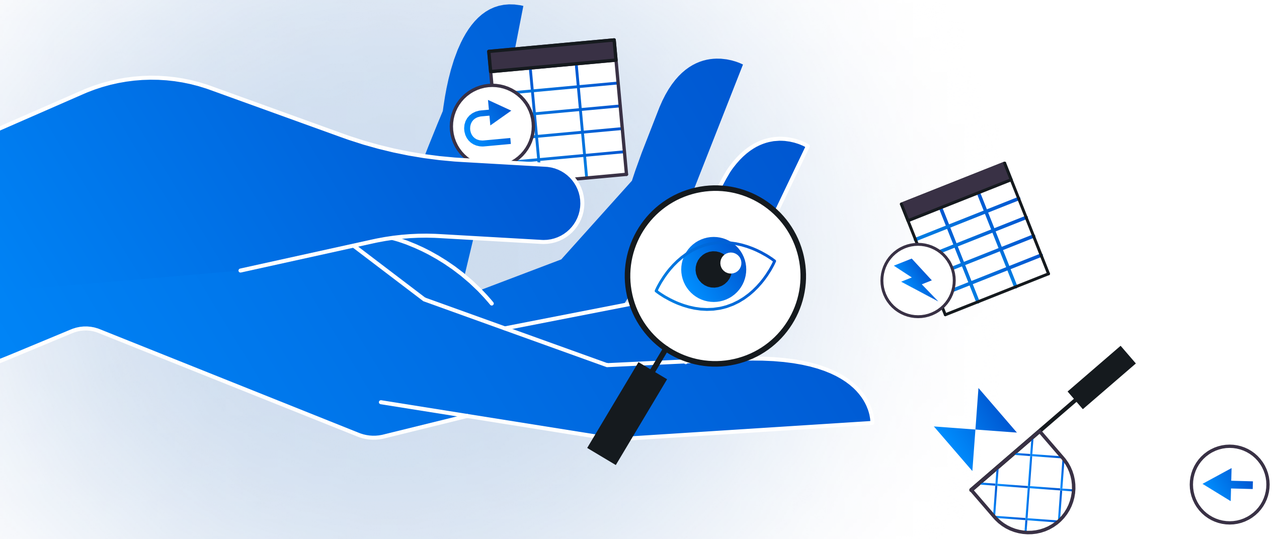
```
-- This will remove the hyphen in @sakila-customer:

SELECT email,
       REPLACE (email, '-', '') AS NEWEMAIL
  FROM public.customer;
```

5. REPLACE can be used with UPDATE to store replaced texts

```
 -- Update the address table by replacing '1923 Hanoi Way' with '1913/2 Hanoi Way':

 UPDATE address
    SET address = REPLACE(address,'1923 Hanoi Way','1913/2 Hanoi Way')
  WHERE postal_code = '35200';
```



SQL UPDATE with JOIN

3 tips for running SQL UPDATE
with JOIN in PostgreSQL

1. Preview records using the SELECT statement

```
-- View the records you need to update using the SELECT
statement. This way, you will see the records prior to the
actual update.

SELECT sta.store_id,
       CAST(CAST(AVG(CAST(sto.Rating AS DECIMAL(3,2))) AS
       DECIMAL(3,2)) AS varchar(9)) AS AverageRating
  INTO ComputedRatings
  FROM staff sta
       LEFT JOIN store sto
         ON sta.store_id = sto.TitleID
GROUP BY sta.store_id;

--UPDATE staff

--SET OverallUserRating = sto.AverageRating

SELECT sta.store_id, sta.OverallUserRating, sto.AverageRating
  FROM staff sta
       INNER JOIN ComputedRatings b
          ON sta.store_id = sto.TitleID;
```

2. Update data using variables

```
-- If you need to use your data recurringly, write it to a
variable and refer to this variable whenever it is required.

SELECT sta.store_id,
       CAST(CAST(AVG(CAST(sto.Rating AS DECIMAL(3,2))) AS
       DECIMAL(3,2)) AS varchar(9)) AS AverageRating
  INTO ComputedRatings
  FROM staff sta
       LEFT JOIN store sto
         ON sta.store_id = sto.TitleID
 GROUP BY sta.store_id;

SELECT * INTO tmpstaff FROM staff;

UPDATE tmpstaff
   SET OverallUserRating = sto.AverageRating
  FROM tmpstaff a
       INNER JOIN ComputedRatings b
          ON sta.store_id = sto.TitleID;
```

3. Use the RETURNING CLAUSE to update data

```
SELECT sta.store_id,
       CAST(CAST(AVG(CAST(sto.Rating AS DECIMAL(3,2))) AS
       DECIMAL(3,2)) AS varchar(9)) AS AverageRating
  INTO ComputedRatings
  FROM staff sta
       LEFT JOIN store sto
         ON sta.store_id = sto.TitleID
 GROUP BY sta.store_id;

SELECT * INTO tmpstaff FROM staff;

UPDATE tmpstaff
   SET OverallUserRating = sto.AverageRating
  FROM tmpstaff a JOIN ComputedRatings b
                    ON ststa.store_id = sto.TitleID
RETURNING sta.store_id, sta.OverallUserRating;
```

devart

SQL INSERT INTO SELECT

The easiest ways to handle duplicates

1. Use WHERE NOT IN

```
INSERT INTO postal_codes (city_id, city, postal_code)
SELECT a.city_id, c.city, a.postal_code
  FROM address a
       INNER JOIN city c
          ON a.city_id = c.city_id
 WHERE a.address_id NOT IN (SELECT address_id FROM store);
```

2. Use WHERE NOT EXISTS

```
INSERT INTO actual_cities (city_id,city, address, country_id)
SELECT a.city_id, c.city, a.address, c.country_id
  FROM address a INNER JOIN city c
                    ON a.city_id = c.city_id
 WHERE NOT EXISTS (SELECT  c.city_id
                     FROM city c
                    WHERE a.city_id = c.city_id
                      AND country_id IN (1, 100));
```

3. Use COUNT(*) = 0

```
INSERT INTO actual_cities (city_id, city, address, country_id)
SELECT a.city_id, c.city, a.address, c.country_id
  FROM address a INNER JOIN city c
                    ON a.city_id = c.city_id
                 WHERE country_id IN (1, 100)
                   AND (SELECT COUNT(*)
                          FROM actual_cities
                         WHERE country_id IN (1, 100))= 0;
```

4. Error handling

```
DO $$
BEGIN
 INSERT INTO postal_codes (city_id, city, postal_code)
 SELECT city_id, city, postal_code
   FROM address a  INNER JOIN city c
                     ON a.city_id = c.city_id
  WHERE a.postal_code = NULL;
 EXCEPTION WHEN OTHERS
           THEN RAISE INFO 'Error Name:%',SQLERRM;
                RAISE INFO 'Error State:%', SQLSTATE;
END
$$
```

5. Use INSERT INTO SELECT DISTINCT

```
INSERT INTO postal_codes (postal_code)
SELECT DISTINCT postal_code
  FROM address;
```

6. Use IF NOT EXISTS

```
DO $$
BEGIN IF NOT EXISTS

(SELECT 1 FROM actual_cities ac
  WHERE ac.country_id IN (1, 100)) THEN
 INSERT INTO actual_cities ( city_id,  city, address,
country_id)
 SELECT a.city_id, c.city, a.address, c.country_id
   FROM address a INNER JOIN city c
                    ON a.city_id = c.city_id
  WHERE EXISTS (SELECT c.city_id
                  FROM city c
                 WHERE a.city_id = c.city_id
                   AND country_id IN (1, 100));
  RAISE INFO 'Not exists';
   ELSE RAISE INFO 'Exists';
  END IF;
END
$$
```

SQL DELETE

5 useful DELETE syntax tips for PostgreSQL developers

1. LIMIT

```
DELETE FROM film
      WHERE YEAR (release_year) = 2006

      ORDER BY last_update
      LIMIT 2;
```

2. Delete all records

```
CREATE TABLE payment_backup AS
SELECT * FROM payment;

DELETE FROM payment_backup;
```

3. DELETE with a JOIN

```
DELETE FROM payment USING staff
      WHERE payment.staff_id = staff.staff_id
        AND staff.username =
'Jon_Stephens@sakilastaff.com';
```

4. DELETE with a subquery

```
DELETE FROM film_actor
      WHERE actor_id IN (SELECT actor_id
                         FROM actor
                        WHERE first_name = 'NICK'
                          AND last_name = 'WAHLBERG');
```

5. Conditional DELETE with WHERE clause

```
DELETE FROM payment
      WHERE payment_id = 200;

-- Delete 3 records using IN:

DELETE FROM payment
      WHERE payment_id IN (201, 202, 203);
```

PostgreSQL Index Examples

16 index examples to speed up your SQL queries

1. Create a simple unique index

```
CREATE UNIQUE INDEX ind_phone
    ON address_new(phone);
```

2. Add a UNIQUE index within CREATE TABLE

```
CREATE TABLE IF NOT EXISTS address_new (
  address_id INTEGER NOT NULL PRIMARY KEY,
  address CHARACTER VARYING(50),
  city_id SMALLINT NOT NULL,
  phone CHARACTER VARYING(20) UNIQUE,
  last_update TIMESTAMP WITHOUT TIME ZONE NOT NULL DEFAULT NOW());
```

3. Create an index with included columns

```
CREATE TABLE IF NOT EXISTS address_new (
  address_id INTEGER NOT NULL PRIMARY KEY,
  address CHARACTER VARYING(50),
  city_id SMALLINT NOT NULL,
  phone CHARACTER VARYING(20),
  last_update TIMESTAMP WITHOUT TIME ZONE NOT NULL DEFAULT NOW(),
  UNIQUE (phone)
  INCLUDE (address));
```

4. Use ALTER TABLE...ADD CONSTRAINT

```
CREATE TABLE IF NOT EXISTS address_new (
  address_id INTEGER NOT NULL,
  address CHARACTER VARYING(50),
  city_id SMALLINT NOT NULL,
  phone CHARACTER VARYING(20),
  last_update TIMESTAMP WITHOUT TIME ZONE NOT NULL DEFAULT NOW());

ADD CONSTRAINT address_pk PRIMARY KEY (address_id);
```

5. Create a filtered index

```
CREATE TABLE IF NOT EXISTS address_new (
  address_id INTEGER NOT NULL PRIMARY KEY,
  address CHARACTER VARYING(50),
  city_id SMALLINT NOT NULL,
  phone CHARACTER VARYING(20),
  last_update TIMESTAMP WITHOUT TIME ZONE NOT NULL DEFAULT NOW());

CREATE INDEX ind_phone
    ON address_new (phone) INCLUDE (address, city_id)
 WHERE address_id > 10
    AND city_id = 4;
```

6. Create a GIST index for the spatial data type

```
ALTER TABLE IF EXISTS city ADD COLUMN points POINT;

CREATE INDEX idx ON city USING GIST (points);
```

7. Add a composite index

```
CREATE TABLE IF NOT EXISTS address_new (
  address_id INTEGER NOT NULL PRIMARY KEY,
  address CHARACTER VARYING(50),
  city_id SMALLINT NOT NULL,
  phone CHARACTER VARYING(20),
  last_update TIMESTAMP WITHOUT TIME ZONE NOT NULL DEFAULT NOW());

CREATE INDEX ind_address_phone
    ON address_new ( address, phone);
```

8. Add an index using CREATE INDEX

```
CREATE TABLE IF NOT EXISTS address_new (
  address_id INTEGER NOT NULL PRIMARY KEY,
  address CHARACTER VARYING(50),
  city_id SMALLINT NOT NULL,
  phone CHARACTER VARYING(20),
  last_update TIMESTAMP WITHOUT TIME ZONE NOT NULL DEFAULT NOW());
CREATE INDEX ind_phone ON address_new (phone);
```

9. Add an index within CREATE TABLE using CONSTRAINT

```
CREATE TABLE IF NOT EXISTS prod_order (
  order_id INTEGER NOT NULL,
  product CHARACTER VARYING(50) COLLATE pg_catalog."default",
  price DOUBLE PRECISION NOT NULL,
  amount DOUBLE PRECISION NOT NULL,
  sum_order DOUBLE PRECISION GENERATED ALWAYS AS ((amount * price)) STORED,
  last_update TIMESTAMP WITHOUT TIME ZONE NOT NULL DEFAULT NOW(),
  CONSTRAINT prod_order_pkey PRIMARY KEY (order_id));
```

10. Create an index on a computed column

```
CREATE TABLE IF NOT EXISTS prod_order (
  order_id INTEGER NOT NULL,
  product CHARACTER VARYING(50) COLLATE pg_catalog."default",
  price DOUBLE PRECISION NOT NULL,
  amount DOUBLE PRECISION NOT NULL,
  sum_order DOUBLE PRECISION GENERATED ALWAYS AS ((amount * price)) STORED,
  last_update TIMESTAMP,
  CONSTRAINT prod_order_pkey PRIMARY KEY (order_id));

CREATE INDEX ind_sum ON prod_order (sum_order);
```

11. Add an index with one column

```
CREATE INDEX ind_phone
    ON address_new(phone);
```

12. Create a GIN index to speed up text search

```
CREATE INDEX ind_txt
    ON film USING GIN (to_tsvector('english', description ));
```

13. Create an index on a materialized view

```
CREATE MATERIALIZED VIEW staff_list
  AS SELECT  s.first_name,
             s.last_name,
             s.email,
             s.username,
             ad.address
       FROM staff s INNER JOIN address ad
                       ON s.address_id = ad.address_id;

CREATE INDEX ind
    ON staff_list (username);
```

14. Create a new index—and drop it immediately if it exists

```
CREATE TABLE IF NOT EXISTS address_new (
  address_id INTEGER NOT NULL PRIMARY KEY,
  address CHARACTER VARYING(50),
  city_id SMALLINT NOT NULL,
  phone CHARACTER VARYING(20),
  last_update TIMESTAMP WITHOUT TIME ZONE NOT NULL DEFAULT NOW());

CREATE INDEX IF NOT EXISTS ind_phone ON address_new (phone);
```

15. Create a GIN index to speed up text search

```
CREATE UNIQUE INDEX ind_phone_address
    ON address_new (phone,address);
```

16. Create a unique index on a computed column

```
CREATE UNIQUE INDEX ind_sum
    ON prod_order (sum_order,product);
```

dbForge Studio for PostgreSQL:
your comprehensive IDE for PostgreSQL databases

dbForge Studio is a multifunctional IDE for PostgreSQL, which helps you handle a variety of tasks, including operations with JOINs. With its help, you can instantly get at least 30% more effective and reduce the time you typically spend on your routine work with databases by about 50%.

DOWNLOAD FREE 30-DAY TRIAL

SQL editor
The built-in SQL editor helps you effectively manage your SQL code with smart completion, syntax highlighting, formatting, refactoring, and a slew of other productivity enhancements that let you get a sharper focus on your work.

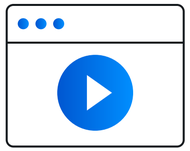
Context-sensitive code completion
With dbForge Studio, you can easily speed up your routine SQL coding by at least 90% with context-sensitive keyword and object suggestions, which include auto-generation of JOIN clauses. Auxiliary features include code snippets, column picker, wildcards, highlighting, and parameter information.

Instant syntax check
The syntax checker is one of the most valuable features integrated into the SQL editor. Whenever the checker detects an error in the code that you are typing, it instantly highlights the problematic place so you can fix it immediately.

SQL formatting
It is easy to improve the readability, consistency, and standardization of your code with the rich SQL formatting options offered by the Studio. Depending on your needs, you can apply automatic, manual, or wizard-aided formatting.

Helpful resources

Here are a few bonuses to help you expand your PostgreSQL skills.



Video tutorials on our YouTube channel

More PostgreSQL insights on our blog

Get in touch with us to request a product demo

Get started with dbForge Studio for free today!

DOWNLOAD FREE 30-DAY TRIAL