insert

- Description
- Parameters
  - Inserting
  - ON CONFLICT Clause
- Outputs
- Notes
- Examples
- Compatibility

INSERT

INSERT

7

SQL - Language Statements

INSERT

create new rows in a table

[WITH [RECURSIVE]

with_query

[, ...]] INSERT INTO

table_name

[AS

alias

] [(

column_name

[, ...])] [OVERRIDING { SYSTEM | USER } VALUE] { DEFAULT VALUES | VALUES ({

expression

| DEFAULT } [, ...]) [, ...] |

query

} [ON CONFLICT [

conflict_target

]

conflict_action

] [RETURNING [WITH ({ OLD | NEW } AS

output_alias

[, ...])] { * |

output_expression

[[AS]

output_name

] } [, ...]]

where

conflict_target

can be one of:

({

index_column_name

| (

index_expression

) } [COLLATE

collation

] [

opclass

] [, ...]) [WHERE

index_predicate

] ON CONSTRAINT

constraint_name

and

conflict_action

is one of:

DO NOTHING DO SELECT [FOR { UPDATE | NO KEY UPDATE | SHARE | KEY SHARE }] [WHERE

condition

] DO UPDATE SET {

column_name

= {

expression

| DEFAULT } | (

column_name

[, ...]) = [ROW] ({

expression

| DEFAULT } [, ...]) | (

column_name

[, ...]) = (

sub-SELECT

) } [, ...] [WHERE

condition

]

Description

`INSERT` inserts new rows into a table. One can insert one or more rows specified by value expressions, or zero or more rows resulting from a query.

The target column names can be listed in any order. If no list of column names is given at all, the default is all the columns of the table in their declared order; or the first <N> column names, if there are only <N> columns supplied by the `VALUES` clause or <query>. The values supplied by the `VALUES` clause or <query> are associated with the explicit or implicit column list left-to-right.

Each column not present in the explicit or implicit column list will be filled with a default value, either its declared default value or null if there is none.

If the expression for any column is not of the correct data type, automatic type conversion will be attempted.

`INSERT` into tables that lack unique indexes will not be blocked by concurrent activity. Tables with unique indexes might block if concurrent sessions perform actions that lock or modify rows matching the unique index values being inserted; the details are covered in ???. `ON CONFLICT` can be used to specify an alternative action to raising a unique constraint or exclusion constraint violation error. (See ON CONFLICT Clause below.)

The optional `RETURNING` clause causes `INSERT` to compute and return value(s) based on each row actually inserted (or selected or updated, if an `ON CONFLICT DO SELECT/UPDATE` clause was used). This is primarily useful for obtaining values that were supplied by defaults, such as a serial sequence number. However, any expression using the table's columns is allowed. The syntax of the `RETURNING` list is identical to that of the output list of `SELECT`. Only rows that were successfully inserted, updated, or selected will be returned. For example, if a row was locked but not updated or selected because an `ON CONFLICT ... WHERE` clause <condition> was not satisfied, the row will not be returned.

You must have `INSERT` privilege on a table in order to insert into it. If `ON CONFLICT DO UPDATE` is present, `UPDATE` privilege on the table is also required. If `ON CONFLICT DO SELECT` is present, `SELECT` privilege on the table is required. If `ON CONFLICT DO SELECT FOR UPDATE/SHARE` is used, `UPDATE` privilege is required on at least one column, in addition to `SELECT` privilege.

If a column list is specified, you only need `INSERT` privilege on the listed columns. Similarly, when `ON CONFLICT DO UPDATE` is specified, you only need `UPDATE` privilege on the column(s) that are listed to be updated. However, all forms of `ON CONFLICT` also require `SELECT` privilege on any column

whose values are read. This includes any column mentioned in <conflict_target> (including columns referred to by the arbiter constraint), and any column mentioned in an `ON CONFLICT DO UPDATE` <expression>, or a `WHERE` clause <condition>.

Use of the `RETURNING` clause requires `SELECT` privilege on all columns mentioned in `RETURNING`. If you use the <query> clause to insert rows from a query, you of course need to have `SELECT` privilege on any table or column used in the query.

Parameters

Inserting

This section covers parameters that may be used when only inserting new rows. Parameters *exclusively* used with the `ON CONFLICT` clause are described separately.

<with_query>
The `WITH` clause allows you to specify one or more subqueries that can be referenced by name in the `INSERT` query. See ??? and ??? for details.

It is possible for the <query> (`SELECT` statement) to also contain a `WITH` clause. In such a case both sets of <with_query> can be referenced within the <query>, but the second one takes precedence since it is more closely nested.

<table_name>
The name (optionally schema-qualified) of an existing table.

<alias>
A substitute name for <table_name>. When an alias is provided, it completely hides the actual name of the table. This is particularly useful when `ON CONFLICT DO UPDATE` targets a table named `excluded`, since that will otherwise be taken as the name of the special table representing the row proposed for insertion.

<column_name>
The name of a column in the table named by <table_name>. The column name can be qualified with a subfield name or array subscript, if needed. (Inserting into only some fields of a composite column leaves the other fields null.) When referencing a column with `ON CONFLICT DO UPDATE`, do not include the table's name in the specification of a target column. For example, `INSERT INTO table_name ... ON CONFLICT DO UPDATE SET table_name.col = 1` is invalid (this follows the general behavior for `UPDATE`).

`OVERRIDING SYSTEM VALUE`
If this clause is specified, then any values supplied for identity columns will override the default sequence-generated values.

For an identity column defined as `GENERATED ALWAYS`, it is an error to insert an explicit value (other than `DEFAULT`) without specifying either `OVERRIDING SYSTEM VALUE` or `OVERRIDING USER VALUE`. (For an identity column defined as `GENERATED BY DEFAULT`, `OVERRIDING SYSTEM VALUE` is the normal behavior and specifying it does nothing, but PostgreSQL allows it as an extension.)

`OVERRIDING USER VALUE`
If this clause is specified, then any values supplied for identity columns are ignored and the default sequence-generated values are applied.

This clause is useful for example when copying values between tables. Writing `INSERT INTO tbl2 OVERRIDING USER VALUE SELECT * FROM tbl1` will copy from `tbl1` all columns that are not identity columns in `tbl2` while values for the identity columns in `tbl2` will be generated by the sequences associated with `tbl2`.

`DEFAULT VALUES`
All columns will be filled with their default values, as if `DEFAULT` were explicitly specified for each column. (An `OVERRIDING` clause is not permitted in this form.)

<expression>
An expression or value to assign to the corresponding column.

`DEFAULT`
The corresponding column will be filled with its default value. An identity column will be filled with a new value generated by the associated sequence. For a generated column, specifying this is permitted but merely specifies the normal behavior of computing the column from its generation expression.

<query>
A query (`SELECT` statement) that supplies the rows to be inserted. Refer to the ??? statement for a description of the syntax.

<output_alias>
An optional substitute name for `OLD` or `NEW` rows in the `RETURNING` list.

By default, old values from the target table can be returned by writing `OLD.column_name` or `OLD.*`, and new values can be returned by writing `NEW.column_name` or `NEW.*`. When an alias is provided, these names are hidden and the old or new rows must be referred to using the alias. For example `RETURNING WITH (OLD AS o, NEW AS n) o.*, n.*`.

<output_expression>
An expression to be computed and returned by the `INSERT` command after each row is inserted or updated. The expression can use any column names of the table named by <table_name>. Write `*`

to return all columns of the inserted or updated row(s).

A column name or `*` may be qualified using `OLD` or `NEW`, or the corresponding <output_alias> for `OLD` or `NEW`, to cause old or new values to be returned. An unqualified column name, or `*`, or a column name or `*` qualified using the target table name or alias will return new values.

For a simple `INSERT`, all old values will be `NULL`. However, for an `INSERT` with an `ON CONFLICT DO SELECT/UPDATE` clause, the old values may be non-`NULL` (when the row proposed for insertion conflicts with an existing row). If the `SELECT` path is taken, the new values will be identical to the old values, since no modification takes place.

<output_name>
A name to use for a returned column.

`ON CONFLICT` Clause

UPSERT

ON CONFLICT

The optional `ON CONFLICT` clause specifies an alternative action to raising a unique violation or exclusion constraint violation error. For each individual row proposed for insertion, either the insertion proceeds, or, if an *arbiter* constraint or index specified by `conflict_target` is violated, the alternative `conflict_action` is taken. `ON CONFLICT DO NOTHING` simply avoids inserting a row as its alternative action. `ON CONFLICT DO UPDATE` updates the existing row that conflicts with the row proposed for insertion as its alternative action. `ON CONFLICT DO SELECT` returns the existing row that conflicts with the row proposed for insertion, optionally with row-level locking.

`conflict_target` can perform *unique index inference*. When performing inference, it consists of one or more <index_column_name> columns and/or <index_expression> expressions, and an optional <index_predicate>. All <table_name> unique indexes that, without regard to order, contain exactly the `conflict_target`-specified columns/expressions are inferred (chosen) as arbiter indexes. If an <index_predicate> is specified, it must, as a further requirement for inference, satisfy arbiter indexes. Note that this means a non-partial unique index (a unique index without a predicate) will be inferred (and thus used by `ON CONFLICT`) if such an index satisfying every other criteria is available. If an attempt at inference is unsuccessful, an error is raised.

`ON CONFLICT DO UPDATE` guarantees an atomic `INSERT` or `UPDATE` outcome; provided there is no independent error, one of those two outcomes is guaranteed, even under high concurrency. This is also known as UPSERT — “UPDATE or INSERT”.

`ON CONFLICT DO SELECT` similarly allows an atomic `INSERT` or `SELECT` outcome. This is also known as idempotent insert or get or create. For `ON CONFLICT DO SELECT`, a `RETURNING` clause *must* be provided.

<conflict_target>
Specifies which conflicts `ON CONFLICT` takes the alternative action on by choosing arbiter indexes. Either performs *unique index inference*, or names a constraint explicitly. For `ON CONFLICT DO NOTHING`, it is optional to specify a `conflict_target`; when omitted, conflicts with all usable constraints (and unique indexes) are handled. For `ON CONFLICT DO UPDATE` and `ON CONFLICT DO SELECT`, a `conflict_target` *must* be provided.

<conflict_action>
`conflict_action` specifies an alternative `ON CONFLICT` action. It can be `DO NOTHING`, a `DO SELECT` clause that allows conflicting rows to be returned, or a `DO UPDATE` clause specifying the exact details of the `UPDATE` action to be performed in case of a conflict.

The `SET` clause in `DO UPDATE` and the `WHERE` clause in both `DO SELECT` and `DO UPDATE` have access to the existing row using the table's name (or an alias), and to the row proposed for insertion using the special `excluded` table. `SELECT` privilege is required on any column in the target table where corresponding `excluded` columns are read.

Note that the effects of all per-row `BEFORE INSERT` triggers are reflected in `excluded` values, since those effects may have contributed to the row being excluded from insertion.

<index_column_name>
The name of a <table_name> column. Used to infer arbiter indexes. Follows `CREATE INDEX` format. `SELECT` privilege on <index_column_name> is required.

<index_expression>
Similar to <index_column_name>, but used to infer expressions on <table_name> columns appearing within index definitions (not simple columns). Follows `CREATE INDEX` format. `SELECT` privilege on any column appearing within <index_expression> is required.

<collation>
When specified, mandates that corresponding <index_column_name> or <index_expression> use a particular collation in order to be matched during inference. Typically this is omitted, as collations usually do not affect whether or not a constraint violation occurs. Follows `CREATE INDEX` format.

<opclass>
When specified, mandates that corresponding <index_column_name> or <index_expression> use particular operator class in order to be matched during inference. Typically this is omitted, as the *equality* semantics are often equivalent across a type's operator classes anyway, or because it's sufficient to trust that the defined unique indexes have the pertinent definition of equality. Follows `CREATE INDEX` format.

<index_predicate>
Used to allow inference of partial unique indexes. Any indexes that satisfy the predicate (which need not actually be partial indexes) can be inferred. Follows `CREATE INDEX` format. `SELECT` privilege on any column appearing within <index_predicate> is required.

<constraint_name>
Explicitly specifies an arbiter *constraint* by name, rather than inferring a constraint or index.

`FOR UPDATE`; `FOR NO KEY UPDATE`; `FOR SHARE`; `FOR KEY SHARE`
When specified in an `ON CONFLICT DO SELECT` clause, conflicting table rows are locked against concurrent updates. See ??? in the ??? documentation.

<condition>
An expression that returns a value of type `boolean`. Only rows for which this expression returns `true` will be updated or selected for return, although all conflicting rows will be locked when `ON CONFLICT DO UPDATE` or `ON CONFLICT DO SELECT FOR UPDATE/SHARE` is specified. Note that <condition> is evaluated last, after a conflict has been identified as a candidate to update or select.

Note that exclusion constraints are not supported as arbiters with `ON CONFLICT DO SELECT/UPDATE`. In all cases, only `NOT DEFERRABLE` constraints and unique indexes are supported as arbiters.

`INSERT` with an `ON CONFLICT DO UPDATE` clause is a “deterministic” statement. This means that the command will not be allowed to affect any single existing row more than once; a cardinality violation error will be raised when this situation arises. Rows proposed for insertion should not duplicate each other in terms of attributes constrained by an arbiter index or constraint.

Note that it is currently not supported for the `ON CONFLICT DO UPDATE` clause of an `INSERT` applied to a partitioned table to update the partition key of a conflicting row such that it requires the row be moved to a new partition.

> [!TIP] It is often preferable to use unique index inference rather than naming a constraint directly using `ON CONFLICT ON CONSTRAINT` <constraint_name>. Inference will continue to work correctly when the underlying index is replaced by another more or less equivalent index in an overlapping way, for example when using `CREATE UNIQUE INDEX ... CONCURRENTLY` before dropping the index being replaced.

Outputs

On successful completion, an `INSERT` command returns a command tag of the form

```
INSERT oid count
```

The <count> is the number of rows inserted, updated, or selected for return. <oid> is always 0 (it used to be the OID assigned to the inserted row if <count> was exactly one and the target table was declared `WITH OIDS` and 0 otherwise, but creating a table `WITH OIDS` is not supported

anymore).

If the `INSERT` command contains a `RETURNING` clause, the result will be similar to that of a `SELECT` statement containing the columns and values defined in the `RETURNING` list, computed over the row(s) affected by the command.

Notes

If the specified table is a partitioned table, each row is routed to the appropriate partition and inserted into it. If the specified table is a partition, an error will occur if one of the input rows violates the partition constraint.

You may also wish to consider using `MERGE`, since that allows mixing `INSERT`, `UPDATE`, and `DELETE` within a single statement. See ???.

Examples

Insert a single row into table films:

```
INSERT INTO films VALUES
    ('UA502', 'Bananas', 105, '1971-07-13', 'Comedy', '82 minutes');
```

In this example, the len column is omitted and therefore it will have the default value:

```
INSERT INTO films (code, title, did, date_prod, kind)
    VALUES ('T_601', 'Yojimbo', 106, '1961-06-16', 'Drama');
```

This example uses the `DEFAULT` clause for the date columns rather than specifying a value:

```
INSERT INTO films VALUES
    ('UA502', 'Bananas', 105, DEFAULT, 'Comedy', '82 minutes');
INSERT INTO films (code, title, did, date_prod, kind)
    VALUES ('T_601', 'Yojimbo', 106, DEFAULT, 'Drama');
```

To insert a row consisting entirely of default values:

```
INSERT INTO films DEFAULT VALUES;
```

To insert multiple rows using the multirow `VALUES` syntax:

```
INSERT INTO films (code, title, did, date_prod, kind) VALUES
    ('B6717', 'Tampopo', 110, '1985-02-10', 'Comedy'),
    ('HG120', 'The Dinner Game', 140, DEFAULT, 'Comedy');
```

This example inserts some rows into table films from a table tmp_films with the same column layout as films:

```
INSERT INTO films SELECT * FROM tmp_films WHERE date_prod < '2004-05-07';
```

This example inserts into array columns:

```
-- Create an empty 3x3 gameboard for noughts-and-crosses
INSERT INTO tictactoe (game, board[1:3][1:3])
    VALUES (1, '{{" "," "," "},{" "," "," "},{" "," "," "}}');
-- The subscripts in the above example aren't really needed
INSERT INTO tictactoe (game, board)
    VALUES (2, '{{X," "," "},{" ",O," "},{" ",X," "}}');
```

Insert a single row into table distributors, returning the sequence number generated by the `DEFAULT` clause:

```
INSERT INTO distributors (did, dname) VALUES (DEFAULT, 'XYZ Widgets')
   RETURNING did;
```

Increment the sales count of the salesperson who manages the account for Acme Corporation, and record the whole updated row along with current time in a log table:

```
WITH upd AS (
  UPDATE employees SET sales_count = sales_count + 1 WHERE id =
    (SELECT sales_person FROM accounts WHERE name = 'Acme Corporation')
    RETURNING *
)
INSERT INTO employees_log SELECT *, current_timestamp FROM upd;
```

Insert or update new distributors as appropriate. Assumes a unique index has been defined that constrains values appearing in the did column. Note that the special excluded table is used to reference values originally proposed for insertion:

```
INSERT INTO distributors (did, dname)
    VALUES (5, 'Gizmo Transglobal'), (6, 'Associated Computing, Inc')
    ON CONFLICT (did) DO UPDATE SET dname = EXCLUDED.dname;
```

Insert or update new distributors as above, returning information about any existing values that were updated, together with the new data inserted. Note that the returned values for old_did and old_dname will be `NULL` for non-conflicting rows:

```
INSERT INTO distributors (did, dname)
    VALUES (5, 'Gizmo Transglobal'), (6, 'Associated Computing, Inc')
    ON CONFLICT (did) DO UPDATE SET dname = EXCLUDED.dname
    RETURNING old.did AS old_did, old.dname AS old_dname,
              new.did AS new_did, new.dname AS new_dname;
```

Insert a distributor, or do nothing for rows proposed for insertion when an existing, excluded row (a row with a matching constrained column or columns after before row insert triggers fire) exists. Example assumes a unique index has been defined that constrains values appearing in the did column:

```
INSERT INTO distributors (did, dname) VALUES (7, 'Redline GmbH')
    ON CONFLICT (did) DO NOTHING;
```

Insert or update new distributors as appropriate. Example assumes a unique index has been defined that constrains values appearing in the did column. `WHERE` clause is used to limit the rows actually updated (any existing row not updated will still be locked, though):

```
-- Don't update existing distributors based in a certain ZIP code
INSERT INTO distributors AS d (did, dname) VALUES (8, 'Anvil Distribution')
    ON CONFLICT (did) DO UPDATE
    SET dname = EXCLUDED.dname || ' (formerly ' || d.dname || ')'
    WHERE d.zipcode <> '21201';

-- Name a constraint directly in the statement (uses associated
-- index to arbitrate taking the DO NOTHING action)
INSERT INTO distributors (did, dname) VALUES (9, 'Antwerp Design')
    ON CONFLICT ON CONSTRAINT distributors_pkey DO NOTHING;
```

Insert new distributor if possible, otherwise return the existing distributor row. Example assumes a unique index has been defined that constrains values appearing in the `did` column. This is useful for get-or-create patterns:

```
INSERT INTO distributors (did, dname) VALUES (11, 'Global Electronics')
    ON CONFLICT (did) DO SELECT
    RETURNING *;
```

Insert a new distributor if the ID doesn't match, otherwise return the existing row, if its name doesn't match:

```
INSERT INTO distributors AS d (did, dname) VALUES (12, 'Micro Devices Inc')
    ON CONFLICT (did) DO SELECT WHERE d.dname != EXCLUDED.dname
    RETURNING *;
```

Insert a new distributor or return and lock the existing row for update. This is useful when you need to ensure exclusive access to the row:

```
INSERT INTO distributors (did, dname) VALUES (13, 'Advanced Systems')
    ON CONFLICT (did) DO SELECT FOR UPDATE
    RETURNING *;
```

Insert new distributor if possible; otherwise `DO NOTHING`. Example assumes a unique index has been defined that constrains values appearing in the did column on a subset of rows where the is_active Boolean column evaluates to `true`:

```
-- This statement could infer a partial unique index on "did"
-- with a predicate of "WHERE is_active", but it could also
-- just use a regular unique constraint on "did"
INSERT INTO distributors (did, dname) VALUES (10, 'Conrad International')
    ON CONFLICT (did) WHERE is_active DO NOTHING;
```

Compatibility

`INSERT` conforms to the SQL standard, except that the `RETURNING` clause is a PostgreSQL extension, as is the ability to use `WITH` with `INSERT`, and the ability to specify an alternative action with `ON CONFLICT`. Also, the case in which a column name list is omitted, but not all the columns are filled from the `VALUES` clause or <query>, is disallowed by the standard. If you prefer a more SQL standard conforming statement than `ON CONFLICT`, see ???.

The SQL standard specifies that `OVERRIDING SYSTEM VALUE` can only be specified if an identity column that is generated always exists. PostgreSQL allows the clause in any case and ignores it if it is not applicable.

Possible limitations of the <query> clause are documented under ???.

□□□ #2
□ □□□ □□□ 2026-10-31 14:12:28 UTC
□ □□□ □□□ 2026-10-31 14:12:31 UTC