update

- Description
- Parameters
- Outputs
- Notes
- Examples
- Compatibility

UPDATE

UPDATE

7

SQL - Language Statements

UPDATE

update rows of a table

[WITH [RECURSIVE]

with_query

[, ...]] UPDATE [ONLY]

table_name

[*] [[AS]

alias

] SET {

column_name

= {

expression

| DEFAULT } | (

column_name

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

expression

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

column_name

[, ...]) = (

sub-SELECT

) } [, ...] [FROM

from_item

[, ...]] [WHERE

condition

| WHERE CURRENT OF

cursor_name

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

output_alias

[, ...])] { * |

output_expression

[[AS]

output_name

] } [, ...]]

Description

`UPDATE` changes the values of the specified columns in all rows that satisfy the condition. Only the columns to be modified need be mentioned in the `SET` clause; columns not explicitly modified retain their previous values.

There are two ways to modify a table using information contained in other tables in the database: using sub-selects, or specifying additional tables in the `FROM` clause. Which technique is more appropriate depends on the specific circumstances.

The optional `RETURNING` clause causes `UPDATE` to compute and return value(s) based on each row actually updated. Any expression using the table's columns, and/or columns of other tables mentioned in `FROM`, can be computed. By default, the new (post-update) values of the table's columns are used, but it is also possible to request the old (pre-update) values. The syntax of the `RETURNING` list is identical to that of the output list of `SELECT`.

You must have the `UPDATE` privilege on the table, or at least on the column(s) that are listed to be updated. You must also have the `SELECT` privilege on any column whose values are read in the <expressions> or <condition>.

Parameters

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

<table_name>
The name (optionally schema-qualified) of the table to update. If `ONLY` is specified before the table name, matching rows are updated in the named table only. If `ONLY` is not specified, matching rows are also updated in any tables inheriting from the named table. Optionally, `*` can be specified after the table name to explicitly indicate that descendant tables are included.

<alias>
A substitute name for the target table. When an alias is provided, it completely hides the actual name of the table. For example, given `UPDATE foo AS f`, the remainder of the `UPDATE` statement must refer to this table as `f` not `foo`.

<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. Do not include the table's name in the specification of a target column — for example, `UPDATE table_name SET table_name.col = 1` is invalid.

<expression>
An expression to assign to the column. The expression can use the old values of this and other columns in the table.

`DEFAULT`
Set the column to its default value (which will be NULL if no specific default expression has been assigned to it). An identity column will be set to 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.

<sub-SELECT>
A `SELECT` sub-query that produces as many output columns as are listed in the parenthesized column list preceding it. The sub-query must yield no more than one row when executed. If it yields one row, its column values are assigned to the target columns; if it yields no rows, NULL values are assigned to the target columns. The sub-query can refer to old values of the current row of the table being updated.

<from_item>
A table expression allowing columns from other tables to appear in the `WHERE` condition and update expressions. This uses the same syntax as the `FROM` clause of a `SELECT` statement; for example, an alias for the table name can be specified. Do not repeat the target table as a <from_item> unless you intend a self-join (in which case it must appear with an alias in the <from_item>).

<condition>
An expression that returns a value of type `boolean`. Only rows for which this expression returns `true` will be updated.

<cursor_name>
The name of the cursor to use in a `WHERE CURRENT OF` condition. The row to be updated is the one most recently fetched from this cursor. The cursor must be a non-grouping query on the `UPDATE`'s target table. Note that `WHERE CURRENT OF` cannot be specified together with a Boolean condition. See ??? for more information about using cursors with `WHERE CURRENT OF`.

<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 `UPDATE` command after each row is updated. The expression can use any column names of the table named by <table_name> or table(s) listed in `FROM`. Write `*` to return all columns.

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.

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

Outputs

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

```
UPDATE count
```

The <count> is the number of rows updated, including matched rows whose values did not change. Note that the number may be less than the number of rows that matched the <condition> when updates were suppressed by a `BEFORE UPDATE` trigger. If <count> is 0, no rows were updated by the query (this is not considered an error).

If the `UPDATE` 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) updated by the command.

Notes

When a `FROM` clause is present, what essentially happens is that the target table is joined to the tables mentioned in the <from_item> list, and each output row of the join represents an update operation for the target table. When using `FROM` you should ensure that the join produces at most one output row for each row to be modified. In other words, a target row shouldn't join to more than one row from the other table(s). If it does, then only one of the join rows will be used to update the target row, but which one will be used is not readily predictable.

Because of this indeterminacy, referencing other tables only within sub-selects is safer, though often harder to read and slower than using a join.

In the case of a partitioned table, updating a row might cause it to no longer satisfy the partition constraint of the containing partition. In that case, if there is some other partition in the partition tree for which this row satisfies its partition constraint, then the row is moved to that partition. If there is no such partition, an error will occur. Behind the scenes, the row movement is actually a `DELETE` and `INSERT` operation.

There is a possibility that a concurrent `UPDATE` or `DELETE` on the row being moved will get a serialization failure error. Suppose session 1 is performing an `UPDATE` on a partition key, and meanwhile a concurrent session 2 for which this row is visible performs an `UPDATE` or `DELETE` operation on this row. In such case, session 2's `UPDATE` or `DELETE` will detect the row movement and raise a serialization failure error (which always returns with an SQLSTATE code '40001').

Applications may wish to retry the transaction if this occurs. In the usual case where the table is not partitioned, or where there is no row movement, session 2 would have identified the newly updated row and carried out the `UPDATE`/`DELETE` on this new row version.

Note that while rows can be moved from local partitions to a foreign-table partition (provided the foreign data wrapper supports tuple routing), they cannot be moved from a foreign-table partition to another partition.

An attempt of moving a row from one partition to another will fail if a foreign key is found to directly reference an ancestor of the source partition that is not the same as the ancestor that's mentioned in the `UPDATE` query.

Examples

Change the word `Drama` to `Dramatic` in the column kind of the table films:

```
UPDATE films SET kind = 'Dramatic' WHERE kind = 'Drama';
```

Adjust temperature entries and reset precipitation to its default value in one row of the table weather:

```
UPDATE weather SET temp_lo = temp_lo+1, temp_hi = temp_lo+15, prcp = DEFAULT
  WHERE city = 'San Francisco' AND date = '2003-07-03';
```

Perform the same operation and return the updated entries, and the old precipitation value:

```
UPDATE weather SET temp_lo = temp_lo+1, temp_hi = temp_lo+15, prcp = DEFAULT
  WHERE city = 'San Francisco' AND date = '2003-07-03'
  RETURNING temp_lo, temp_hi, prcp, old.prcp AS old_prcp;
```

Use the alternative column-list syntax to do the same update:

```
UPDATE weather SET (temp_lo, temp_hi, prcp) = (temp_lo+1, temp_lo+15, DEFAULT)
  WHERE city = 'San Francisco' AND date = '2003-07-03';
```

Increment the sales count of the salesperson who manages the account for Acme Corporation, using the `FROM` clause syntax:

```
UPDATE employees SET sales_count = sales_count + 1 FROM accounts
  WHERE accounts.name = 'Acme Corporation'
  AND employees.id = accounts.sales_person;
```

Perform the same operation, using a sub-select in the `WHERE` clause:

```
UPDATE employees SET sales_count = sales_count + 1 WHERE id =
  (SELECT sales_person FROM accounts WHERE name = 'Acme Corporation');
```

Update contact names in an accounts table to match the currently assigned salespeople:

```
UPDATE accounts SET (contact_first_name, contact_last_name) =
    (SELECT first_name, last_name FROM employees
     WHERE employees.id = accounts.sales_person);
```

A similar result could be accomplished with a join:

```
UPDATE accounts SET contact_first_name = first_name,
                    contact_last_name = last_name
  FROM employees WHERE employees.id = accounts.sales_person;
```

However, the second query may give unexpected results if employees.id is not a unique key, whereas the first query is guaranteed to raise an error if there are multiple id matches. Also, if there is no match for a particular accounts.sales_person entry, the first query will set the corresponding name fields to NULL, whereas the second query will not update that row at all.

Update statistics in a summary table to match the current data:

```
UPDATE summary s SET (sum_x, sum_y, avg_x, avg_y) =
    (SELECT sum(x), sum(y), avg(x), avg(y) FROM data d
     WHERE d.group_id = s.group_id);
```

Attempt to insert a new stock item along with the quantity of stock. If the item already exists, instead update the stock count of the existing item. To do this without failing the entire transaction, use savepoints:

```
BEGIN;
-- other operations
SAVEPOINT sp1;
INSERT INTO wines VALUES('Chateau Lafite 2003', '24');
-- Assume the above fails because of a unique key violation,
-- so now we issue these commands:
ROLLBACK TO sp1;
UPDATE wines SET stock = stock + 24 WHERE winename = 'Chateau Lafite 2003';
-- continue with other operations, and eventually
COMMIT;
```

Change the kind column of the table films in the row on which the cursor `c_films` is currently positioned:

```
UPDATE films SET kind = 'Dramatic' WHERE CURRENT OF c_films;
```

Updates affecting many rows can have negative effects on system performance, such as table bloat, increased replica lag, and increased lock contention. In such situations it can make sense to perform the operation in smaller batches, possibly with a `VACUUM` operation on the table between batches. While there is no `LIMIT` clause for `UPDATE`, it is possible to get a similar effect through the use of a Common Table Expression and a self-join. With the standard PostgreSQL table access method, a self-join on the system column ctid is very efficient:

```
WITH exceeded_max_retries AS (
  SELECT w.ctid FROM work_item AS w
    WHERE w.status = 'active' AND w.num_retries > 10
    ORDER BY w.retry_timestamp
    FOR UPDATE
    LIMIT 5000
)
UPDATE work_item SET status = 'failed'
  FROM exceeded_max_retries AS emr
  WHERE work_item.ctid = emr.ctid;
```

This command will need to be repeated until no rows remain to be updated. (This use of ctid is only safe because the query is repeatedly run, avoiding the problem of changed ctids.) Use of an `ORDER BY` clause allows the command to prioritize which rows will be updated; it can also prevent deadlock with other update operations if they use the same ordering. If lock contention is a concern, then `SKIP LOCKED` can be added to the CTE to prevent multiple commands from updating the same row. However, then a final `UPDATE` without `SKIP LOCKED` or `LIMIT` will be needed to ensure that no matching rows were overlooked.

Compatibility

This command conforms to the SQL standard, except that the `FROM` and `RETURNING` clauses are PostgreSQL extensions, as is the ability to use `WITH` with `UPDATE`.

Some other database systems offer a `FROM` option in which the target table is supposed to be listed again within `FROM`. That is not how PostgreSQL interprets `FROM`. Be careful when porting applications that use this extension.

According to the standard, the source value for a parenthesized sub-list of target column names can be any row-valued expression yielding the correct number of columns. PostgreSQL only allows

the source value to be a row constructor or a sub-`SELECT`. An individual column's updated value can be specified as `DEFAULT` in the row-constructor case, but not inside a sub-`SELECT`.

□□□ #2
□ □□□ □□□ 2026-10-31 14:14:59 UTC
□ □□□ □□□ 2026-10-31 14:15:00 UTC