# The MERGE Command

Álvaro Herrera
PostgreSQL developer


https://www.EnterpriseDB.com/

Prague PostgreSQL Developer Day
February 2023

# So what is MERGE anyway?

- A "new" SQL DML command

**ISO/IEC 9075-2:2016(E)**
**14.12 <merge statement>**

## 14.12 <merge statement>

*This Subclause is modified by Subclause 14.5, "<merge statement>", in ISO/IEC 9075-14.*

### Function

Conditionally update and/or delete rows of a table and/or insert new rows into a table.

- Introduced in SQL:2003
- Is it UPSERT?

# So what is MERGE anyway?

- A "new" SQL DML command

**ISO/IEC 9075-2:2016(E)**
**14.12 <merge statement>**

## 14.12 <merge statement>

*This Subclause is modified by Subclause 14.5, "<merge statement>", in ISO/IEC 9075-14.*

### Function

Conditionally update and/or delete rows of a table and/or insert new rows into a table.

- Introduced in SQL:2003
- Is it UPSERT?

# What MERGE is not

- MERGE is not UPSERT
- (PostgreSQL ended up introducing non-standard INSERT ON CONFLICT UPDATE in 9.5 (2016)

```
INSERT INTO wines (winery, brand, variety, year, bottles)
     VALUES ('Concha y Toro', 'Sunrise', 'Chardonnay', 2021, 96)
ON CONFLICT (winery, brand, variety, year) DO
 UPDATE SET bottles = wines.bottles + EXCLUDED.bottles;
```

- 👍 Special (index-)row locking is used
- 👍 Fast and "concurrency correct"
- 👎 It can't do DELETE
- 👎 It is not standard and very different from MERGE
- 👎 Concurrency considerations are different
- )

# What MERGE is not

- MERGE is not UPSERT
- (PostgreSQL ended up introducing non-standard INSERT ON CONFLICT UPDATE in 9.5 (2016)

```
INSERT INTO wines (winery, brand, variety, year, bottles)
     VALUES ('Concha y Toro', 'Sunrise', 'Chardonnay', 2021, 96)
ON CONFLICT (winery, brand, variety, year) DO
 UPDATE SET bottles = wines.bottles + EXCLUDED.bottles;
```

- 👍 Special (index-)row locking is used
- 👍 Fast and “concurrency correct”
- 👎 It can’t do DELETE
- 👎 It is not standard and very different from MERGE
- 👎 Concurrency considerations are different
- )

# What MERGE is not

- MERGE is not UPSERT
- (PostgreSQL ended up introducing non-standard INSERT ON CONFLICT UPDATE in 9.5 (2016)

```
INSERT INTO wines (winery, brand, variety, year, bottles)
     VALUES ('Concha y Toro', 'Sunrise', 'Chardonnay', 2021, 96)
ON CONFLICT (winery, brand, variety, year) DO
 UPDATE SET bottles = wines.bottles + EXCLUDED.bottles;
```

- 👍 Special (index-)row locking is used
- 👍 Fast and “concurrency correct”
- 👎 It can’t do DELETE
- 👎 It is not standard and very different from MERGE
- 👎 Concurrency considerations are different
- )

# MERGE history

- First implementation attempt in PostgreSQL 11 (2018)
- Fully implemented in PostgreSQL 15 (2022)
- Other RDBMS systems have it
  - Oracle™ 11 had it (2007?)
  - SQL Server™ 2008 had it
- ... so Postgres having it, improves chances of migration
  - (even though there are still some differences)

# MERGE history

- First implementation attempt in PostgreSQL 11 (2018)
- Fully implemented in PostgreSQL 15 (2022)
- Other RDBMS systems have it
  - Oracle™ 11 had it (2007?)
  - SQL Server™ 2008 had it
- ... so Postgres having it, improves chances of migration
  - (even though there are still some differences)

# MERGE example

```
CREATE TABLE wines (winery text, brand text, variety text, year int,
                    bottles int);
ALTER TABLE wines ADD UNIQUE (winery, brand, variety, year);
CREATE TABLE shipment (LIKE wines);

INSERT INTO shipment VALUES
       ('Concha y Toro', 'Sunrise', 'Chardonnay', 2021, 96),
       ('Concha y Toro', 'Sunrise', 'Merlot', 2022, 120),
       ('Concha y Toro', 'Marqués de Casa y Concha', 'Carmenere',
                         2021, 48),
       ('Concha y Toro', 'Casillero del Diablo', 'Cabernet Sauvignon',
                         2019, 240);
```

# MERGE example

```
CREATE TABLE wines (winery text, brand text, variety text, year int, bottles int);
CREATE TABLE shipment (LIKE wines);

MERGE INTO wines AS w
     USING shipment AS s
        ON (w.winery, w.brand, w.variety, w.year) =
           (s.winery, s.brand, s.variety, s.year)

WHEN MATCHED THEN
     UPDATE SET bottles = w.bottles + s.bottles

WHEN NOT MATCHED THEN
     INSERT (winery, brand, variety, year, bottles)
     VALUES (s.winery, s.brand, s.variety, s.year, s.bottles);
```

# MERGE syntax

```
[ WITH clause ]

  MERGE INTO target_table
       USING { source table or query }
          ON { join condition }

   WHEN MATCHED [ AND expression ] THEN
       UPDATE SET { columns / values }

   WHEN MATCHED [ AND expression ] THEN
       DELETE

   WHEN NOT MATCHED [ AND expression ] THEN
       INSERT { columns / values }
   ;
```

# Multiple WHEN-clause example

```
MERGE INTO wines w
     USING shipment s
        ON (w.winery, ...) = (s.winery, ...)

 WHEN MATCHED AND (w.bottles + s.bottles) <= 0
     DELETE
 WHEN MATCHED AND (w.bottles + s.bottles) > 1200
     UPDATE SET on_sale = true, bottles = w.bottles+s.bottles
 WHEN MATCHED AND (w.bottles + s.bottles) < 120
     UPDATE SET on_sale = false, bottles = w.bottles+s.bottles
 WHEN MATCHED
     UPDATE SET bottles = w.bottles + s.bottles
 WHEN NOT MATCHED AND s.bottles < 12 THEN
     DO NOTHING
 WHEN NOT MATCHED THEN
     INSERT ( ... ) VALUES ( ... )
```

# Dealing with concurrency

- Easiest is to not run two MERGEs concurrently targetting the same table
  - (perhaps: `LOCK TABLE wines IN SHARE MODE`)
- If you must have two, make them not have WHEN NOT MATCHED THEN INSERT clauses
- If you must allow concurrent insertion, rewrite to INSERT ON CONFLICT UPDATE
- If you do not lock the table, test extensively for concurrent scenarios
  - plan to spend ~~at least twice as long~~ much longer testing than developing

# Dealing with concurrency

- Easiest is to not run two MERGEs concurrently targetting the same table
  - (perhaps: `LOCK TABLE wines IN SHARE MODE`)
- If you must have two, make them not have WHEN NOT MATCHED THEN INSERT clauses
- If you must allow concurrent insertion, rewrite to INSERT ON CONFLICT UPDATE
- If you do not lock the table, test extensively for concurrent scenarios
  - plan to spend ~~at least twice as long~~ much longer testing than developing

# Duplicates in MERGE source

What if my source table has “duplicates”?

```
-- update
ERROR:  MERGE command cannot affect row a second time


-- insert
ERROR:  duplicate key value violates unique constraint "wine
DETAIL:  Key (winery, brand, variety, year)=(Concha y Toro,
```

# Duplicates in MERGE source

What if my source table has “duplicates”?

```
-- update
ERROR:  MERGE command cannot affect row a second time

-- insert
ERROR:  duplicate key value violates unique constraint "wine
DETAIL:  Key (winery, brand, variety, year)=(Concha y Toro,
```

# Duplicates in MERGE source (2)

```
MERGE INTO wines AS w
      USING (   SELECT winery, brand, variety, year,
                       sum(bottles) as bottles
                  FROM shipment
                GROUP BY winery, brand, variety, year
             ) AS s
          ON (w.winery, w.brand, w.variety, w.year) =
             (s.winery, s.brand, s.variety, s.year)
WHEN MATCHED THEN
       UPDATE SET bottles = w.bottles + s.bottles
WHEN NOT MATCHED THEN
       INSERT (winery, brand, variety, year, bottles)
       VALUES (s.winery, s.brand, s.variety, s.year, s.bottles)
;
```

# MERGE example 3

What if I run MERGE twice with the same source?

```
ALTER TABLE shipment ADD COLUMN marked timestamp with time zone;


WITH unmarked_shipment AS
 (UPDATE shipment SET marked = now() WHERE marked IS NULL
         RETURNING winery, brand, variety, year, bottles)

MERGE INTO wines AS w
     USING unmarked_shipment AS s
        ON (w.winery, w.brand, w.variety, w.year) =
           (s.winery, s.brand, s.variety, s.year)
WHEN MATCHED THEN
     UPDATE SET bottles = w.bottles + s.bottles

WHEN NOT MATCHED THEN
     INSERT (winery, brand, variety, year, bottles)
     VALUES (s.winery, s.brand, s.variety, s.year, s.bottles);
```

# MERGE example 3

What if I run MERGE twice with the same source?

```
ALTER TABLE shipment ADD COLUMN marked timestamp with time zone;


WITH unmarked_shipment AS
 (UPDATE shipment SET marked = now() WHERE marked IS NULL
         RETURNING winery, brand, variety, year, bottles)

MERGE INTO wines AS w
     USING unmarked_shipment AS s
        ON (w.winery, w.brand, w.variety, w.year) =
           (s.winery, s.brand, s.variety, s.year)
WHEN MATCHED THEN
     UPDATE SET bottles = w.bottles + s.bottles

WHEN NOT MATCHED THEN
     INSERT (winery, brand, variety, year, bottles)
     VALUES (s.winery, s.brand, s.variety, s.year, s.bottles);
```

EDB

# MERGE example 3

What if I run MERGE twice with the same source?

```
ALTER TABLE shipment ADD COLUMN marked timestamp with time zone;


WITH unmarked_shipment AS
 (UPDATE shipment SET marked = now() WHERE marked IS NULL
         RETURNING winery, brand, variety, year, bottles)

MERGE INTO wines AS w
     USING unmarked_shipment AS s
        ON (w.winery, w.brand, w.variety, w.year) =
           (s.winery, s.brand, s.variety, s.year)
WHEN MATCHED THEN
     UPDATE SET bottles = w.bottles + s.bottles

WHEN NOT MATCHED THEN
     INSERT (winery, brand, variety, year, bottles)
     VALUES (s.winery, s.brand, s.variety, s.year, s.bottles);
```

# Combining both examples

```
WITH unmarked_shipment AS
 (UPDATE shipment SET marked = now() WHERE marked IS NULL
         RETURNING winery, brand, variety, year, bottles)
MERGE INTO wines AS w
     USING (SELECT winery, brand, variety, year,
                          sum(bottles) as bottles
                   FROM unmarked_shipment
                GROUP BY winery, brand, variety, year) AS s
        ON (w.winery, w.brand, w.variety, w.year) =
           (s.winery, s.brand, s.variety, s.year)
WHEN MATCHED THEN
     UPDATE SET bottles = w.bottles + s.bottles
WHEN NOT MATCHED THEN
     INSERT (winery, brand, variety, year, bottles)
     VALUES (s.winery, s.brand, s.variety, s.year, s.bottles)
;
```

# Trigger behavior

- Boring (it behaves as you'd expect)
  - BEFORE EACH STATEMENT
    - INSERT
    - UPDATE
    - DELETE
  - BEFORE EACH ROW
    - Each row as scanned by the join
  - AFTER EACH ROW
    - Each row in the same order as above
  - AFTER EACH STATEMENT
    - DELETE
    - UPDATE
    - INSERT

# FOR STATEMENT triggers – transition table example

```
CREATE TABLE wine_audit (op varchar(1), datetime timestamptz,
                         oldrow jsonb, newrow jsonb);

CREATE FUNCTION wine_audit() RETURNS trigger LANGUAGE plpgsql AS $$
  BEGIN
    IF (TG_OP = 'DELETE') THEN
      INSERT INTO wine_audit
           SELECT 'D', now(), row_to_json(o), NULL FROM old_table o;
    ELSIF (TG_OP = 'INSERT') THEN
      INSERT INTO wine_audit
           SELECT 'I', now(), NULL, row_to_json(n) FROM new_table n;
    ELSIF (TG_OP = 'UPDATE') THEN
      ... -- for later
    END IF;
    RETURN NULL;
  END;
$$;
```

# Transition table – the triggers

```
CREATE TRIGGER wine_update
  AFTER UPDATE ON wines
  REFERENCING OLD TABLE AS old_table NEW TABLE AS new_table
  FOR EACH STATEMENT EXECUTE FUNCTION wine_audit();

CREATE TRIGGER wine_insert
  AFTER INSERT ON wines
  REFERENCING NEW TABLE AS new_table
  FOR EACH STATEMENT EXECUTE FUNCTION wine_audit();

CREATE TRIGGER wine_delete
  AFTER DELETE ON wines
  REFERENCING OLD TABLE AS old_table
  FOR EACH STATEMENT EXECUTE FUNCTION wine_audit();
```

# Transition table – UPDATE part

```
DECLARE
  oldrec record;
  newrec jsonb;
  i      integer := 0;
BEGIN
  FOR oldrec IN SELECT * FROM old_table LOOP
    newrec := row_to_json(n) FROM new_table n OFFSET i LIMIT 1;
    i := i + 1;
    INSERT INTO wine_audit
         SELECT 'U', now(), row_to_json(oldrec), newrec;
  END LOOP;
END;
```

# Transition table – complete example

```
CREATE FUNCTION wine_audit() RETURNS trigger LANGUAGE plpgsql AS $$
  BEGIN
    IF (TG_OP = 'DELETE') THEN
      INSERT INTO wine_audit
          SELECT 'D', now(), row_to_json(o), NULL FROM old_table o;
    ELSIF (TG_OP = 'INSERT') THEN
      INSERT INTO wine_audit
          SELECT 'I', now(), NULL, row_to_json(n) FROM new_table n;
    ELSIF (TG_OP = 'UPDATE') THEN
      DECLARE
        oldrec record;
        newrec jsonb;
        i      integer := 0;
      BEGIN
        FOR oldrec IN SELECT * FROM old_table LOOP
          newrec := row_to_json(n) FROM new_table n OFFSET i LIMIT 1;
          i := i + 1;
          INSERT INTO wine_audit
              SELECT 'U', now(), row_to_json(oldrec), newrec;
        END LOOP;
      END;

    END IF;
    RETURN NULL;
  END;
$$;
```

# Permissions

- Permissions are required according to operations specified
- ... even if at run-time the operations are not executed
- Requirements are like normal INSERT/UPDATE/DELETE

# Possible target types

- Supported
  - Regular tables
  - Partitioned tables
  - Tables with “legacy” inheritance
- Not yet supported
  - Foreign tables
  - Materialized views
  - Updatable views

# Possible target types

- Supported
  - Regular tables
  - Partitioned tables
  - Tables with “legacy” inheritance
- Not yet supported
  - Foreign tables
  - Materialized views
  - Updatable views

# Development Credits

- Simon Riggs
- Pavan Deolasee
- Amit Langote
- Álvaro Herrera

# Future projects

All by Dean Rasheed:

- Support for updatable views
- Support for RETURNING
- Support for WHEN NOT MATCHED BY SOURCE

# Questions

## Questions?

Álvaro Herrera, EDB
alvherre@alvh.no-ip.org
https://alvherre.cl/
Mastodon: https://lile.cl/@alvherre/