Trigger: How it Works in PostgreSQL Internals

PGConf.EU 2022, Berlin, Germany

Oct 26, 2022

Yugo Nagata (SRA OSS LLC)

About me

- Yugo Nagata
  - Software Engineer & Researcher at SRA OSS LLC
  - Research and Development on PostgreSQL
    - Incremental View Maintenance (IVM)
    - pg_ivm (https://github.com/sraoss/pg_ivm)
  - Lecture on PostgreSQL Internal

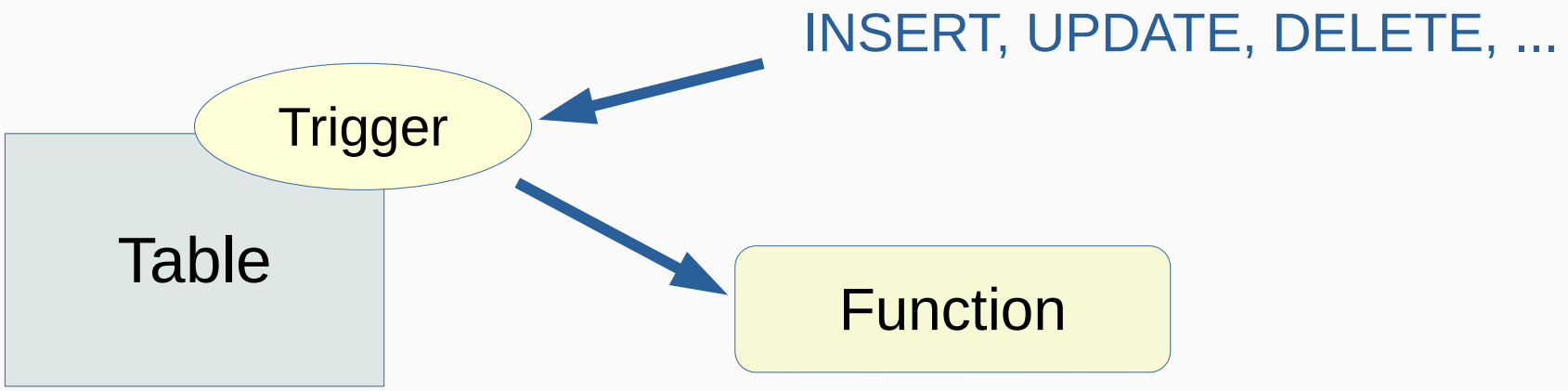
Outline

- Overview of Trigger
- How Triggers are created
- How Triggers work

Overview of Trigger

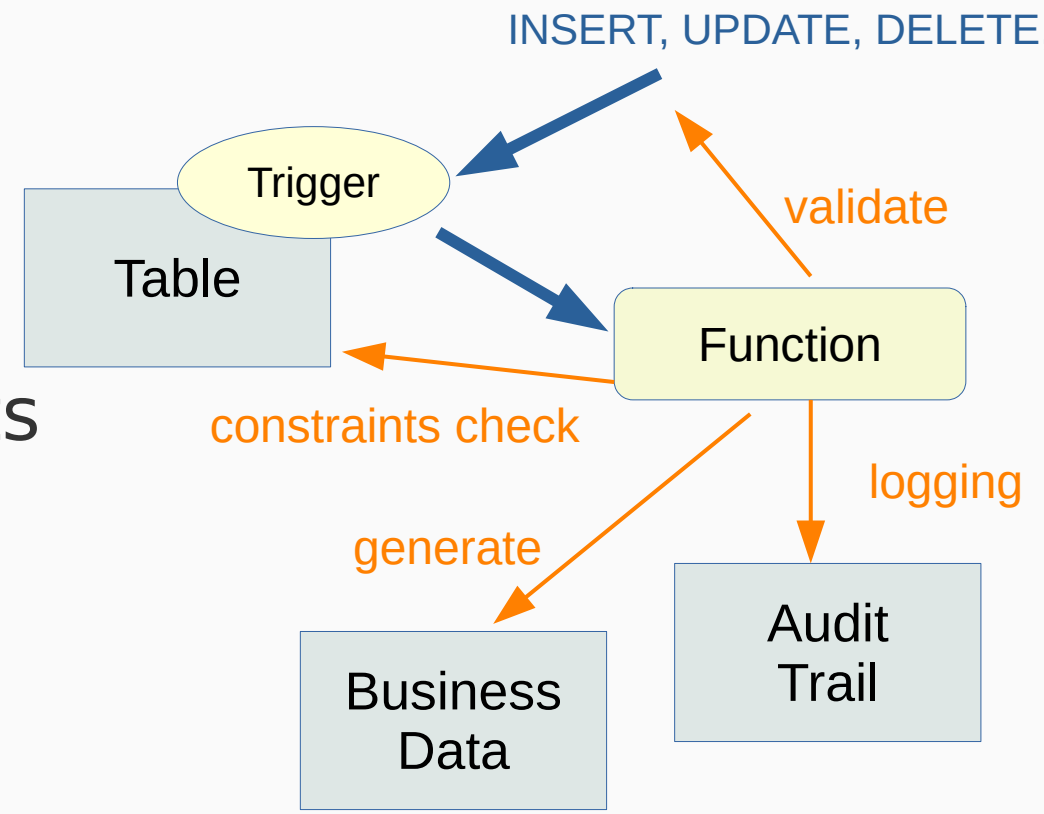
Trigger

- A special function is automatically executed whenever a certain type of operation is performed
- Attached to tables, views, and foreign tables



Use Cases of Triggers

- Audit trail / logging
- Input data validation
- Enforcing / checking constraints
- Complex business rules

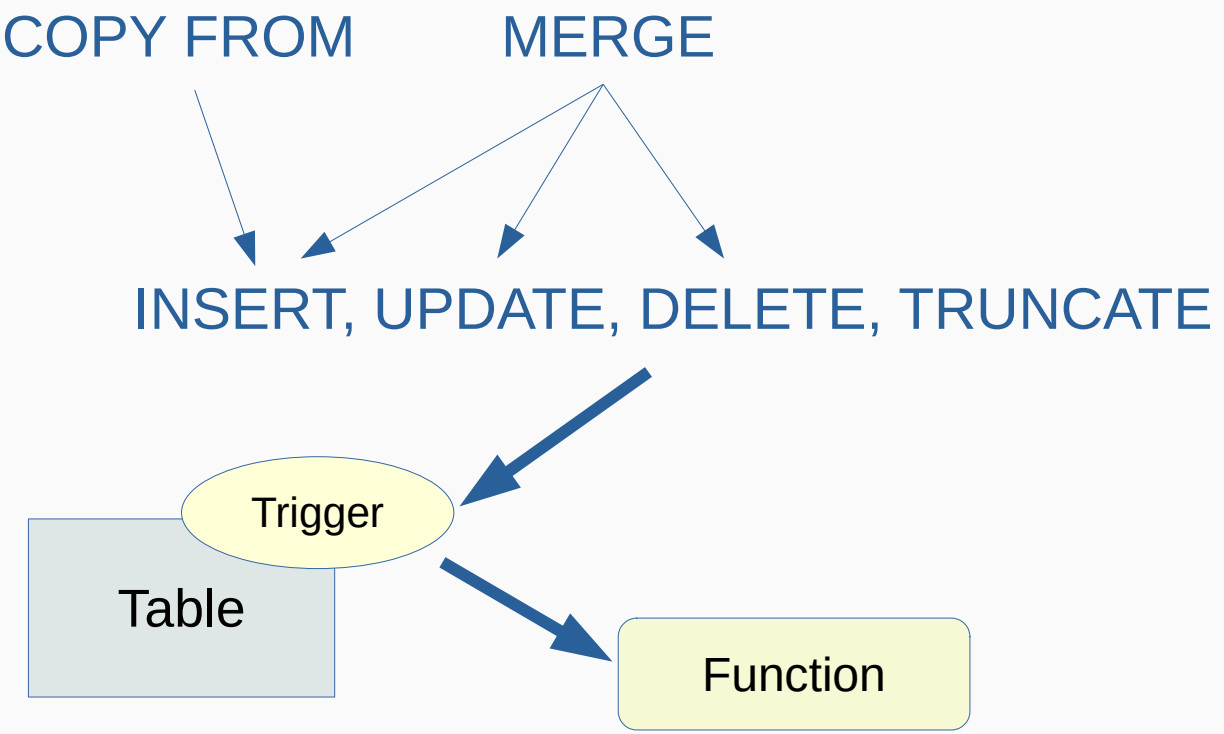


When a Trigger is Fired

After or before ...

- INSERT, UPDATE, DELETE
- TRUNCATE

- COPY FROM
- MERGE (PG15+)



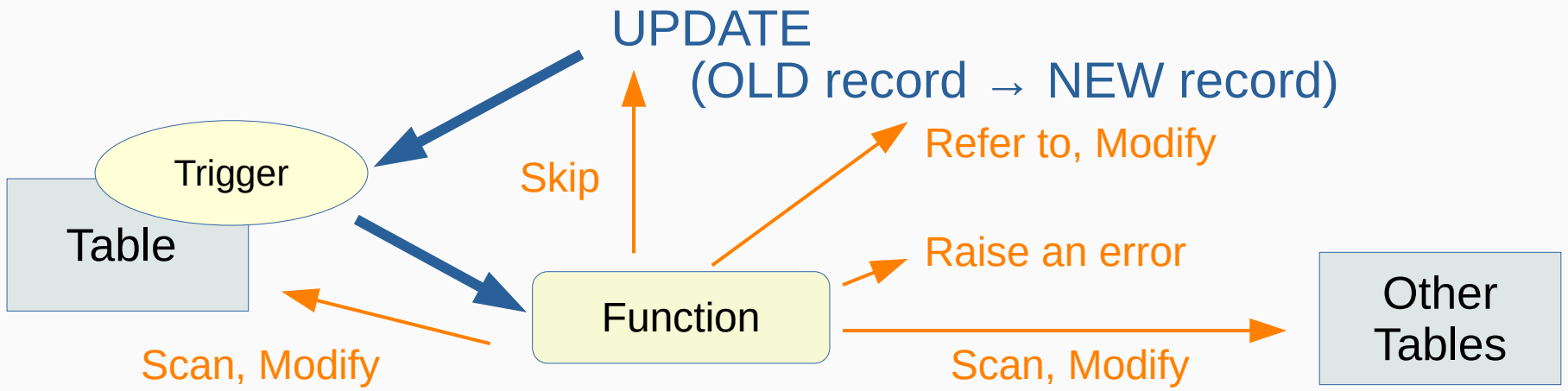
Types of Triggers

- Per-row (row-level) trigger
  - Invoked once for each row that is affected by the statement
- Per-statement (statement-level) trigger
  - Invoked only once when an appropriate statement is executed

- Triggers on TRUNCATE may only be defined at statement level.

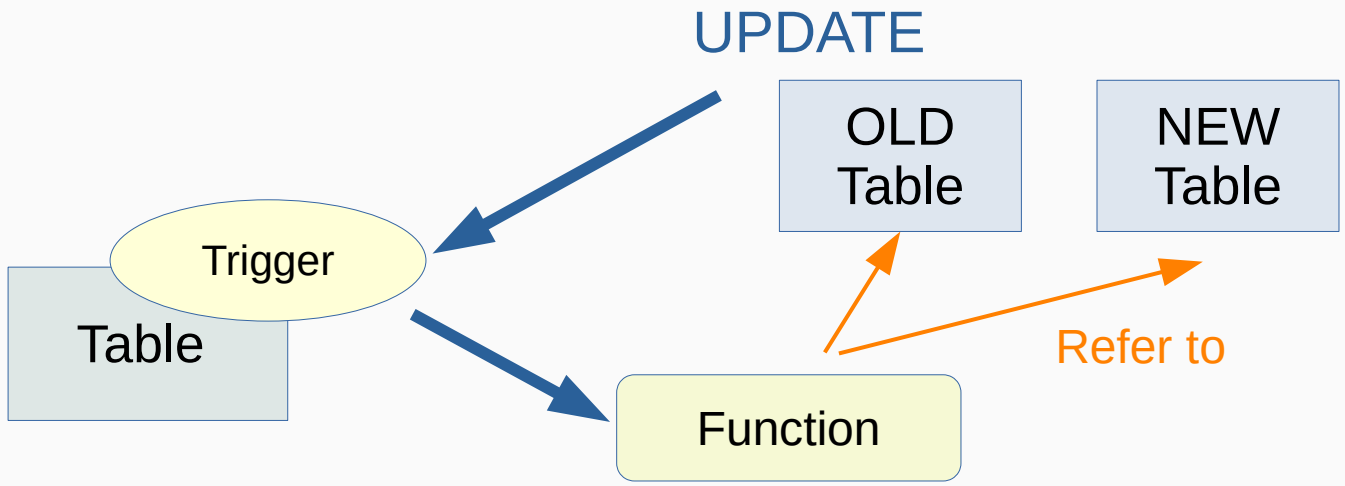
What a Trigger Function Can Do

- Scan and/or modify other tables
- Refer to the old and/or new row record (in row-level triggers)
- Modify the row being inserted or updated, or skip the operation (in row-level BEFORE triggers)



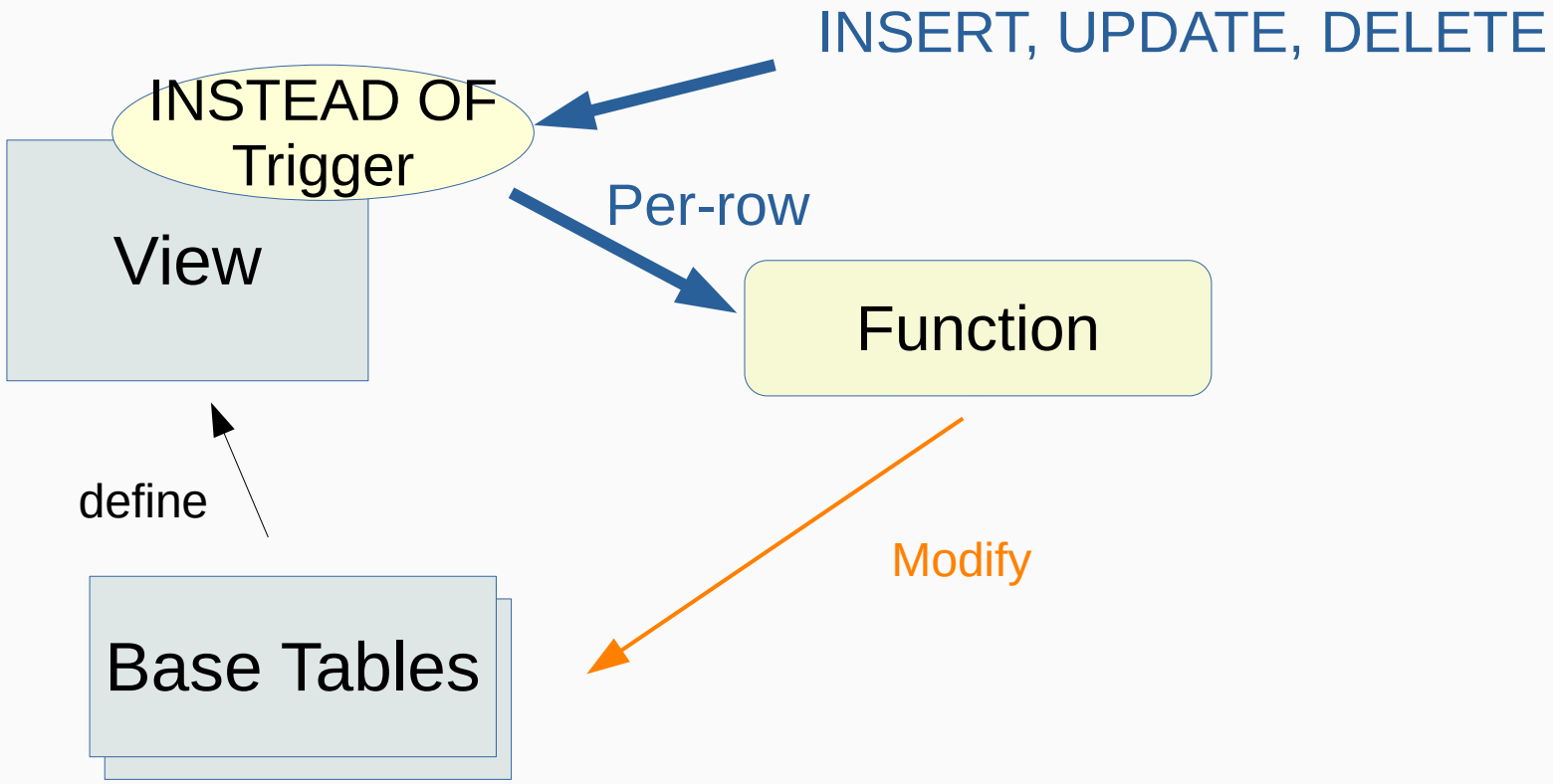
Transition Tables

- Set of affected rows
  - OLD TABLE: before-images of all rows updated or deleted
  - NEW TABLE: after-images of all rows updated or inserted
- In AFTER triggers (both statement-level and row-level)



INSTEAD OF Triggers on Views

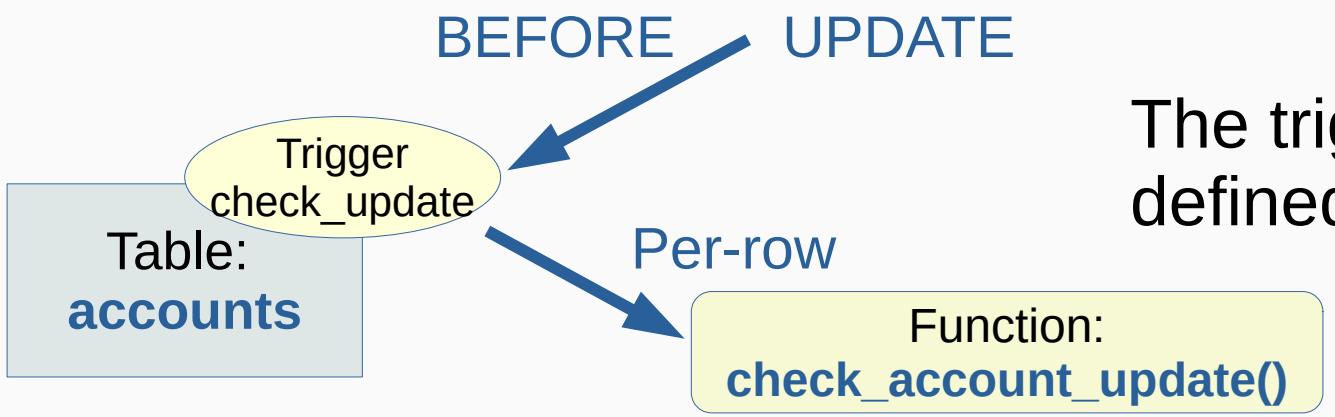
Making a complex view updatable



How Triggers are created

CREATE TRIGGER

```
CREATE TRIGGER check_update
    BEFORE UPDATE ON accounts
    FOR EACH ROW
    EXECUTE FUNCTION check_account_update();
```



The trigger function must be defined before creating a trigger.

Creating a trigger function

```
CREATE FUNCTION check_account_update()
 RETURNS trigger AS $$
  BEGIN
   IF NEW.data > 100 THEN
     RAISE EXCEPTION 'data out of data';
   END IF;
   RETURN NEW;
  END;
 $$ LANGUAGE plpgsql;
```

Returning type: “trigger”
No arguments

Return the row to be inserted,
or the new row after update.

- Can be modified
- NULL means skip of the operation.

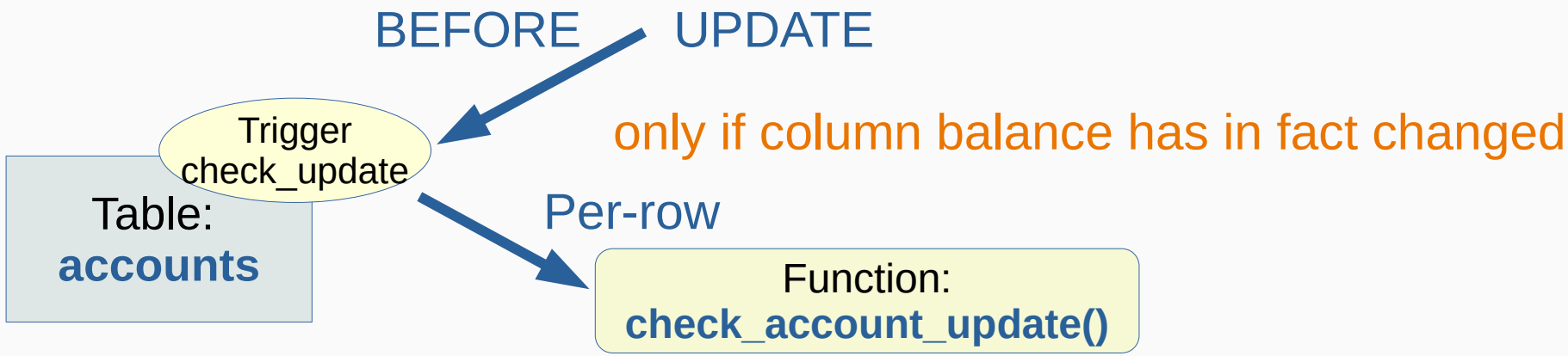
CREATE TRIGGER with arguments

```
CREATE TRIGGER check_update
    BEFORE UPDATE ON accounts
    FOR EACH ROW
    EXECUTE FUNCTION check_account_update(true);
```

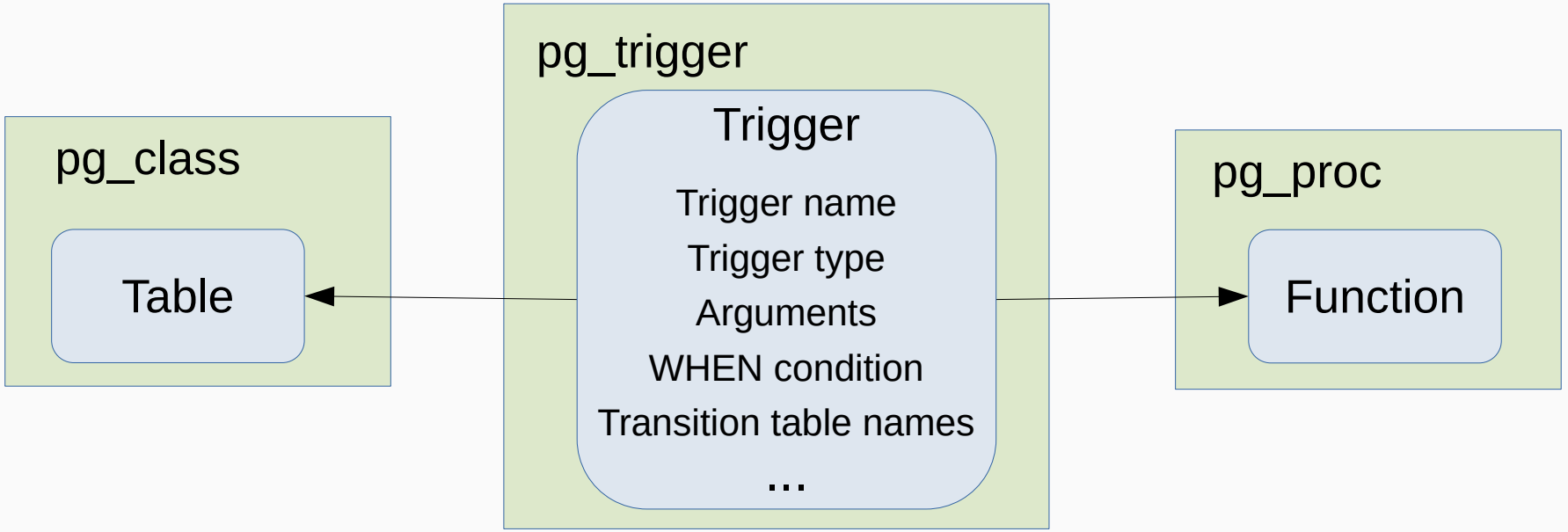
```
CREATE FUNCTION check_account_update()
 RETURNS trigger AS $$
  BEGIN
   IF NEW.data > 100 AND TG_ARGV[0] = 'true' THEN
     RAISE EXCEPTION 'data out of data';
   END IF;
   RETURN NEW;
  END;
 $$ LANGUAGE plpgsql;
```

WHEN condition

```
CREATE TRIGGER check_update
    BEFORE UPDATE ON accounts
    FOR EACH ROW
    WHEN (OLD.balance IS DISTINCT FROM NEW.balance)
    EXECUTE FUNCTION check_account_update();
```

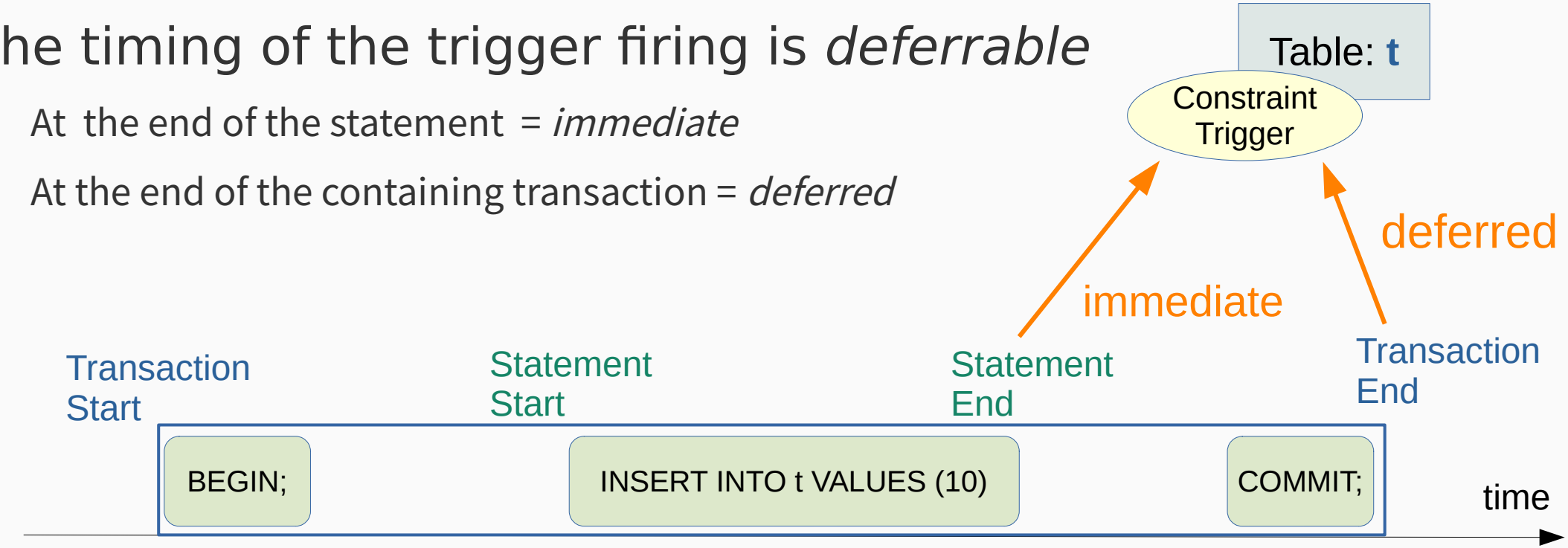


System Catalogs



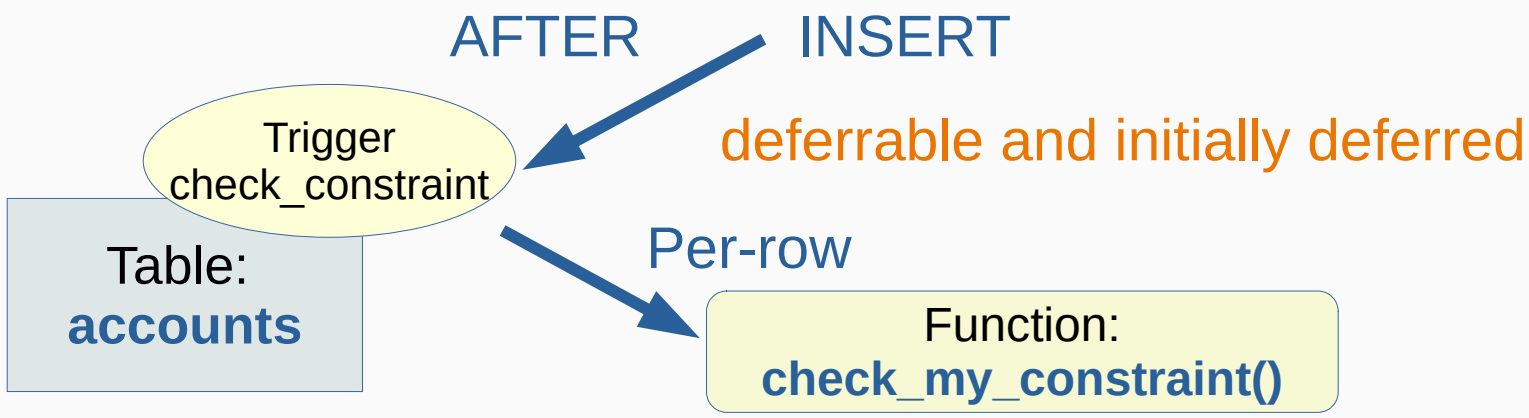
Constraint Trigger

- A trigger for implementing a constraint
- Row-level AFTER trigger only
- The timing of the trigger firing is *deferrable*
  - At the end of the statement = *immediate*
  - At the end of the containing transaction = *deferred*



Creating a constraint trigger

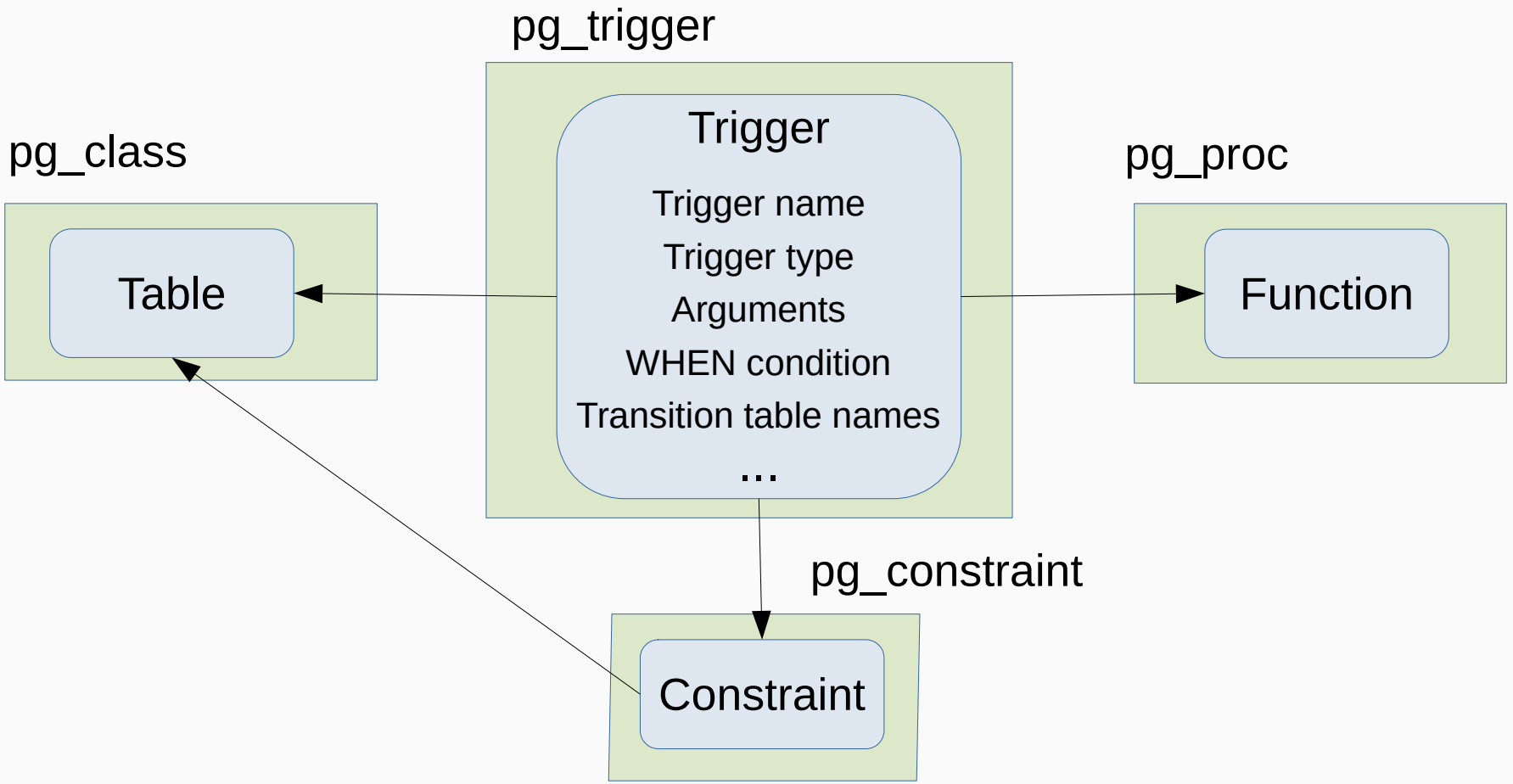
```
CREATE CONSTRAINT TRIGGER check_constraint
    AFTER INSERT ON accounts
    DEFERRABLE INITIALLY DEFERRED
    FOR EACH ROW
    EXECUTE FUNCTION check_my_constraint();
```



SET CONSTRAINTS

```
BEGIN;

SET CONSTRAINTS check_constraint DEFERRED;
 or
SET CONSTRAINTS check_constraint IMMEDIATE;

...

END;
```

System Catalogs



Constraint Triggers Created Internally

- Foreign key constraint (Referential Integrity)

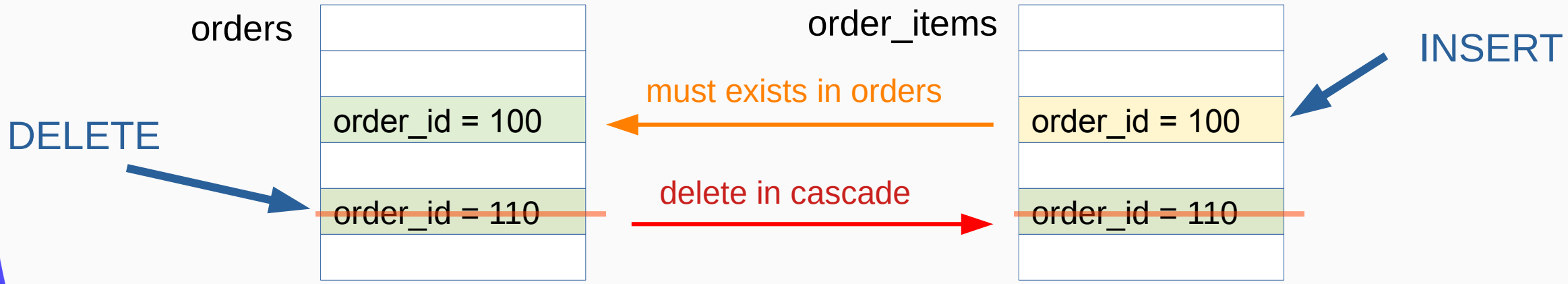
```
CREATE TABLE orders (
    order_id integer PRIMARY KEY,
    product_no integer REFERENCES products (product_no),
    quantity integer
);
```

- Primary key / unique / exclusion constraint (deferrable only)

```
CREATE TABLE distributors (
    did     integer,
    name    varchar(40) UNIQUE DEFERRABLE
);
```

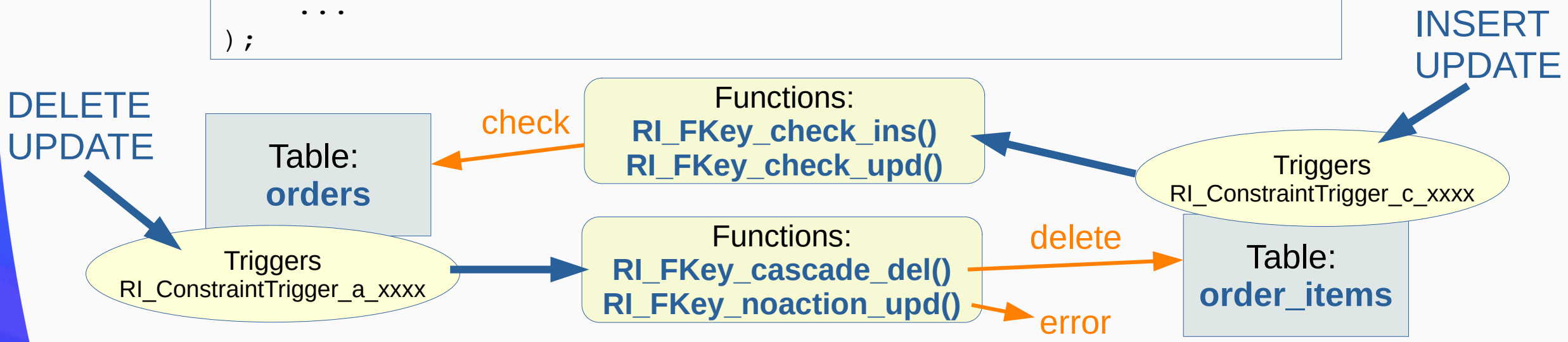
Foreign Key Constraint

```
CREATE TABLE orders (
    order_id integer PRIMARY KEY,
    ...
);

CREATE TABLE order_items (
    order_id integer REFERENCES orders ON DELETE CASCADE,
    ...
);
```



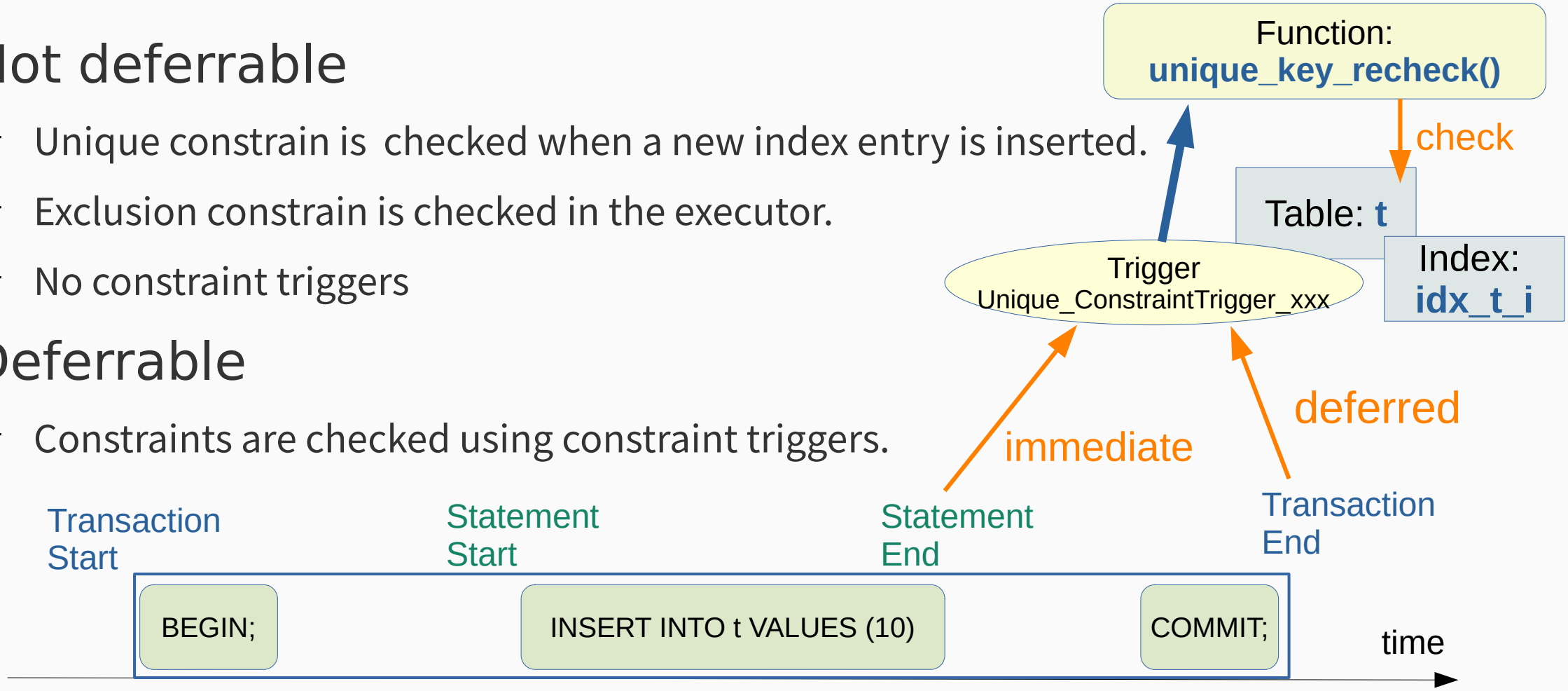
Foreign Key Constraint Triggers

```
CREATE TABLE orders (
    order_id integer PRIMARY KEY,
    ...
);

CREATE TABLE order_items (
    order_id integer REFERENCES orders ON DELETE CASCADE,
    ...
);
```

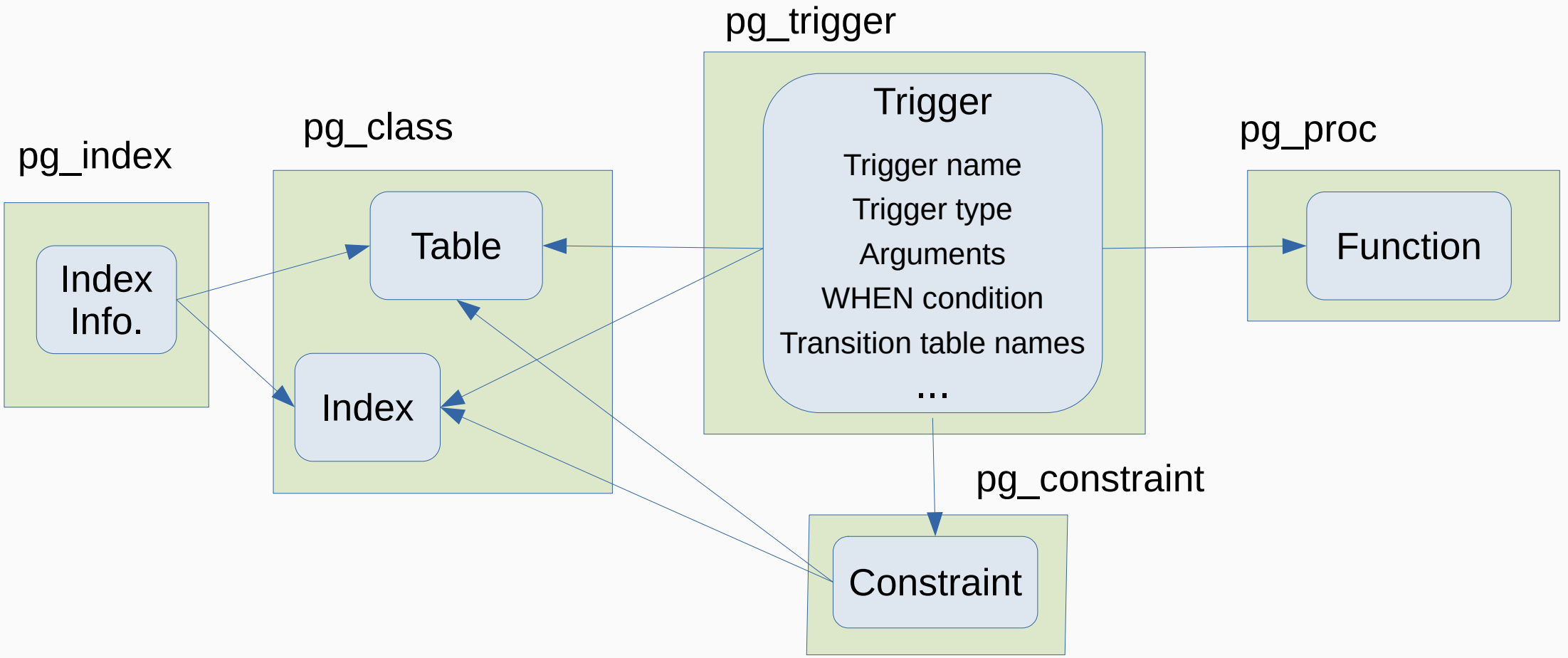


Primary Key / Unique / Exclusion Constraint

- Not deferrable
  - Unique constrain is checked when a new index entry is inserted.
  - Exclusion constrain is checked in the executor.
  - No constraint triggers
- Deferrable
  - Constraints are checked using constraint triggers.

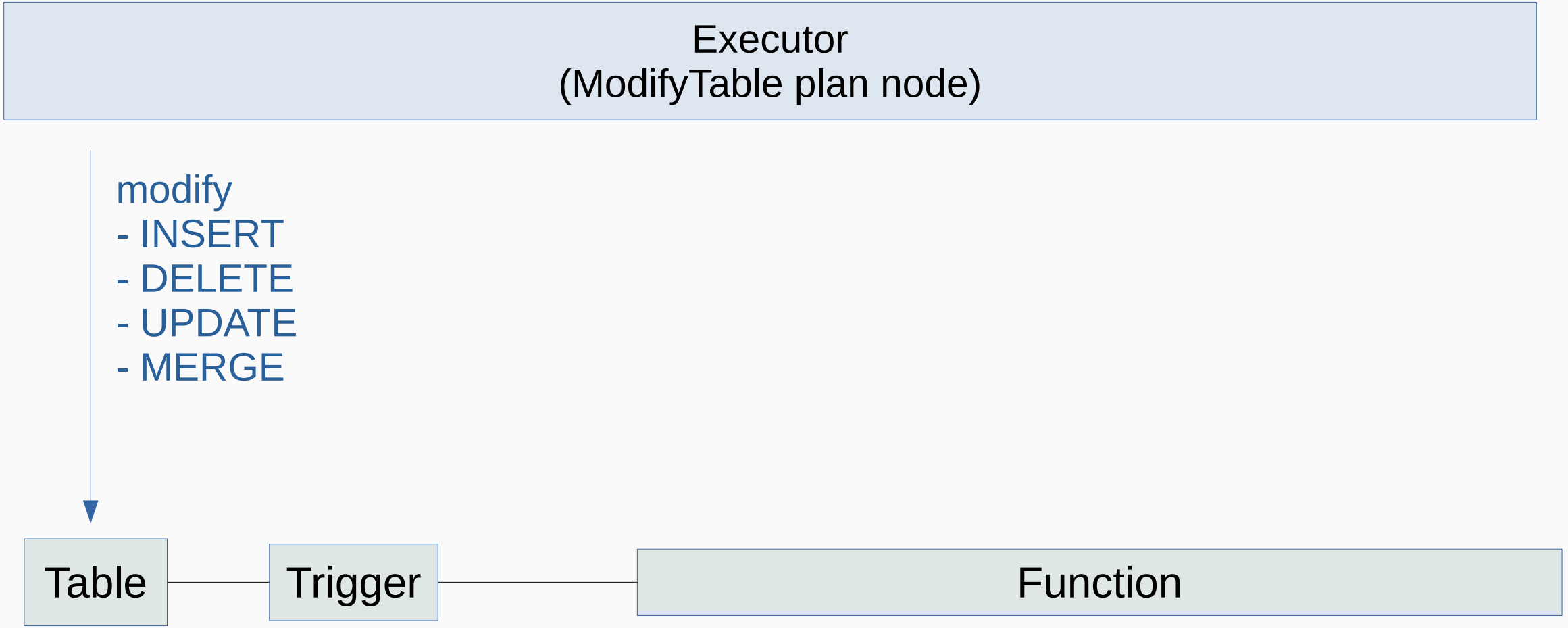


System Catalogs

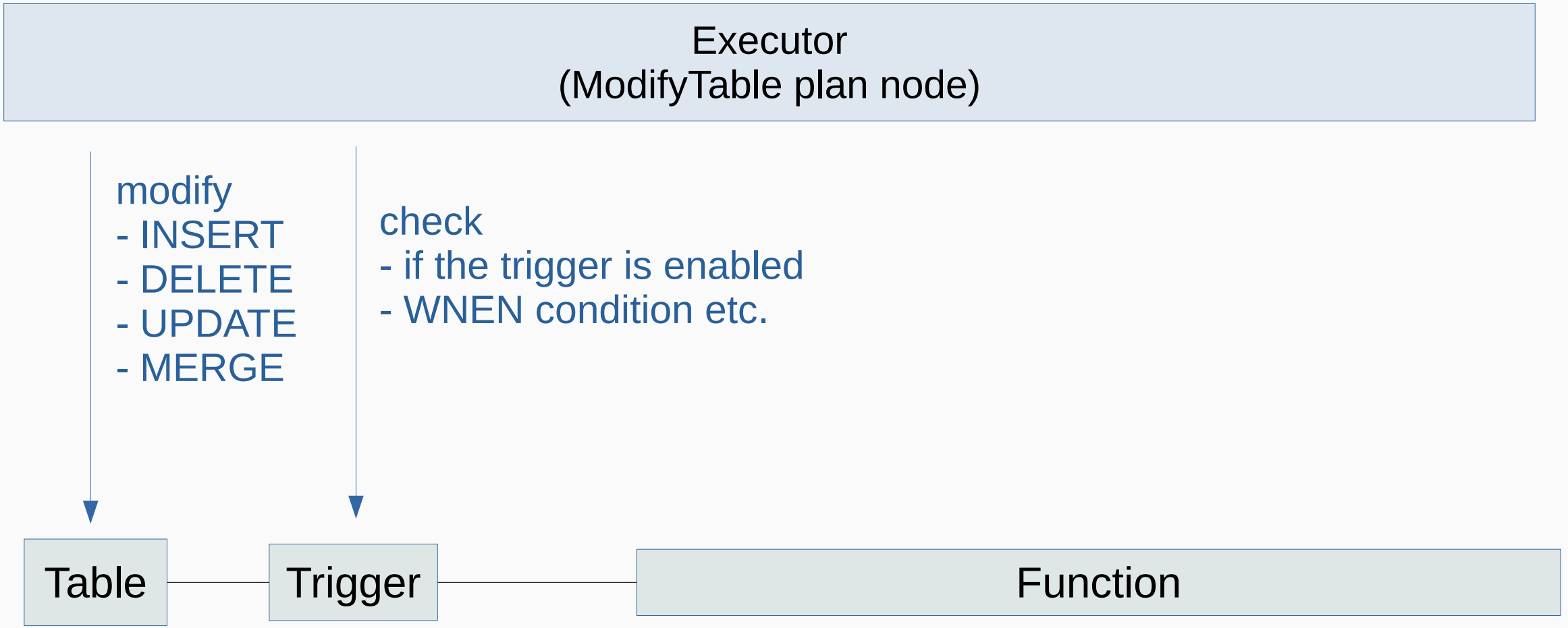


How Triggers Work

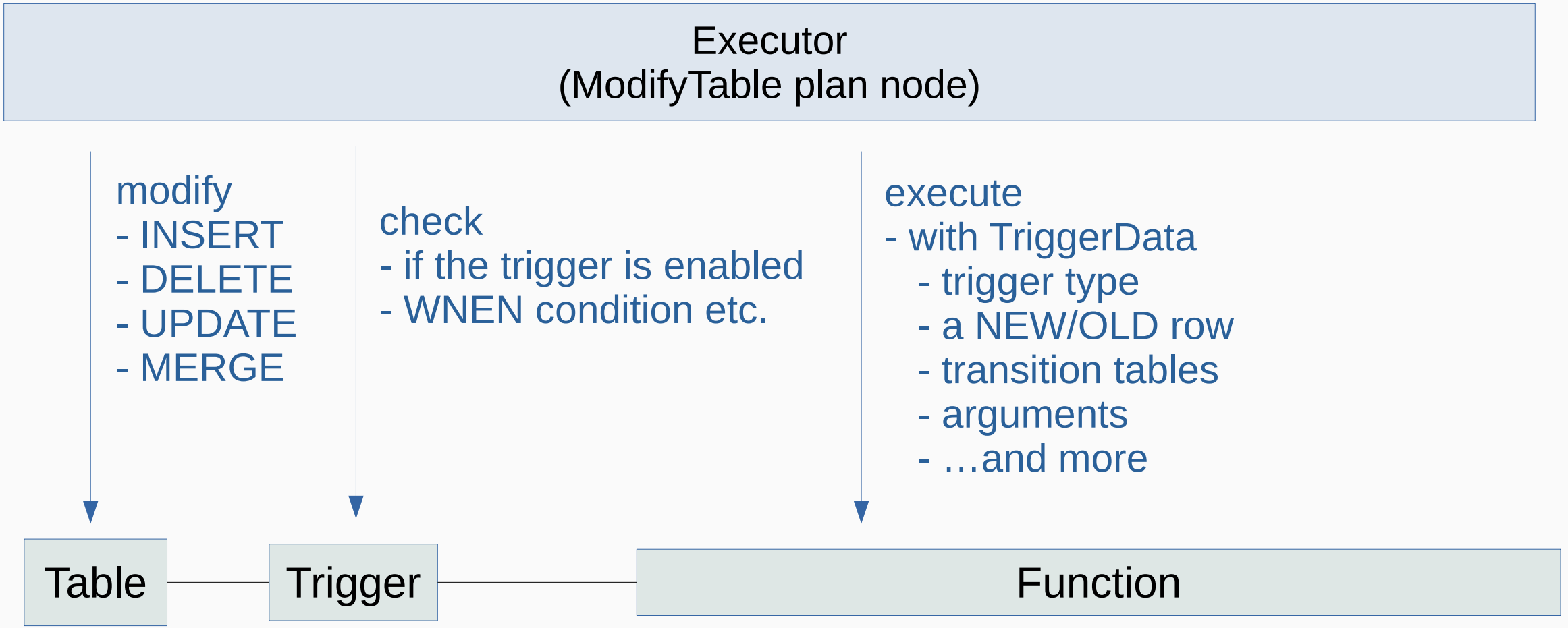
How a trigger is fired



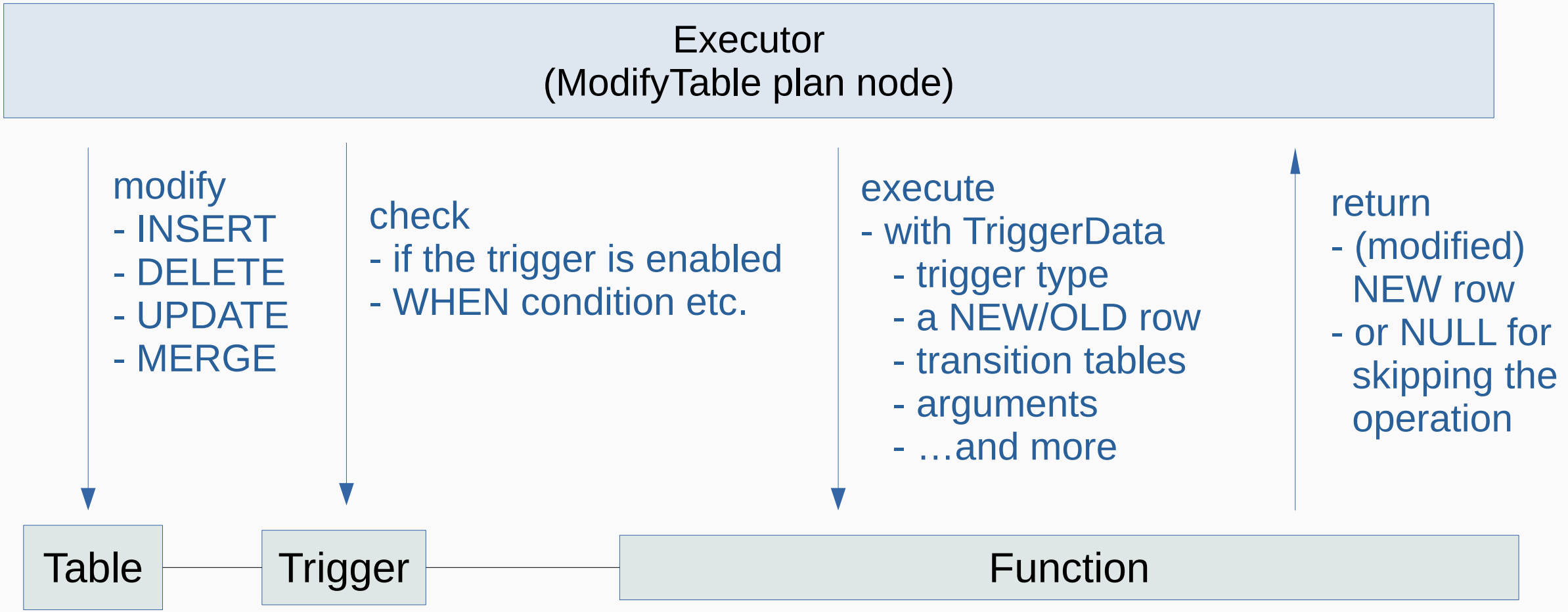
How a trigger is fired



How a trigger is fired



How a trigger is fired



When Triggers are Fired: (1) Statement-level BEFORE triggers

```
UPDATE tbl SET val = val + 100 WHERE flag = 1;
```

tbl

id = 1, val = 10, flag = 0
id = 2, val = 20, flag = 0
id = 3, val = 30, flag = 1
id = 4, val = 40, flag = 1
id = 5, val = 50, flag = 0
id = 6, val = 60, flag = 1
id = 7, val = 70, flag = 0
id = 8, val = 80, flag = 0

Executed Triggers:

Statement-level BEFORE triggers

When Triggers are Fired:
(2) The statement starts processing rows

```
UPDATE tbl SET val = val + 100 WHERE flag = 1;
```

Start

tbl

id = 1, val = 10, flag = 0
id = 2, val = 20, flag = 0
id = 3, val = 30, flag = 1
id = 4, val = 40, flag = 1
id = 5, val = 50, flag = 0
id = 6, val = 60, flag = 1
id = 7, val = 70, flag = 0
id = 8, val = 80, flag = 0

Executed Triggers:

Statement-level BEFORE triggers

When Triggers are Fired: (3) Row-level BEFORE triggers

```
UPDATE tbl SET val = val + 100 WHERE flag = 1;
```

Start

tbl

id = 1, val = 10, flag = 0
id = 2, val = 20, flag = 0
id = 3, val = 30, flag = 1
id = 4, val = 40, flag = 1
id = 5, val = 50, flag = 0
id = 6, val = 60, flag = 1
id = 7, val = 70, flag = 0
id = 8, val = 80, flag = 0

Executed Triggers:

Statement-level BEFORE triggers
Row-level BEFORE triggers:
  for id = 3

When Triggers are Fired: (4) The row is updated

```
UPDATE tbl SET val = val + 100 WHERE flag = 1;
```

Start

tbl

id = 1, val = 10, flag = 0
id = 2, val = 20, flag = 0
id = 3, val = 130, flag = 1
id = 4, val = 40, flag = 1
id = 5, val = 50, flag = 0
id = 6, val = 60, flag = 1
id = 7, val = 70, flag = 0
id = 8, val = 80, flag = 0

Executed Triggers:

Statement-level BEFORE triggers
Row-level BEFORE triggers:
  for id = 3

When Triggers are Fired: (5) Row-level AFTER triggers

```
UPDATE tbl SET val = val + 100 WHERE flag = 1;
```

Start

tbl

id = 1, val = 10, flag = 0
id = 2, val = 20, flag = 0
id = 3, val = 130, flag = 1
id = 4, val = 40, flag = 1
id = 5, val = 50, flag = 0
id = 6, val = 60, flag = 1
id = 7, val = 70, flag = 0
id = 8, val = 80, flag = 0

Executed Triggers:

Statement-level BEFORE triggers
Row-level BEFORE triggers:
  for id = 3

queue:

Row-level AFTER triggers:
  for id = 3

Transition Tables:

OLD table:
(id=3, val=30, flag=1)

NEW table:
(id=3, val=130, flag=1)

When Triggers are Fired: (6) Processing the next rows ...

```
UPDATE tbl SET val = val + 100 WHERE flag = 1;
```

Start

tbl

id = 1, val = 10, flag = 0
id = 2, val = 20, flag = 0
id = 3, val = 130, flag = 1
id = 4, val = 140, flag = 1
id = 5, val = 50, flag = 0
id = 6, val = 160, flag = 1
id = 7, val = 70, flag = 0
id = 8, val = 80, flag = 0

Executed Triggers:

Statement-level BEFORE triggers
Row-level BEFORE triggers:
  for id = 3, 4, 6

queue:

Row-level AFTER triggers:
  for id = 3, 4, 6

Transition Tables:

OLD table:
 (id=3, val=30, flag=1)
 (id=4, val=40, flag=1)
 (id=6, val=60, flag=1)

NEW table:
 (id=3, val=130, flag=1)
 (id=3, val=140, flag=1)
 (id=3, val=160, flag=1)

When Triggers are Fired: (7) Statement-level AFTER triggers

```
UPDATE tbl SET val = val + 100 WHERE flag = 1;
```

Start

tbl

id = 1, val = 10, flag = 0
id = 2, val = 20, flag = 0
id = 3, val = 130, flag = 1
id = 4, val = 140, flag = 1
id = 5, val = 50, flag = 0
id = 6, val = 160, flag = 1
id = 7, val = 70, flag = 0
id = 8, val = 80, flag = 0

End

Executed Triggers:

Statement-level BEFORE triggers
Row-level BEFORE triggers:
  for id = 3, 4, 6

queue:

Row-level AFTER triggers:
  for id = 3, 4, 6
Statement-level AFTER triggers

Transition Tables:

OLD table:
 (id=3, val=30, flag=1)
 (id=4, val=40, flag=1)
 (id=6, val=60, flag=1)

NEW table:
 (id=3, val=130, flag=1)
 (id=3, val=140, flag=1)
 (id=3, val=160, flag=1)

When Triggers are Fired: (8) Queued triggers are executed

```
UPDATE tbl SET val = val + 100 WHERE flag = 1;
```

Start

tbl

id = 1, val = 10, flag = 0
id = 2, val = 20, flag = 0
id = 3, val = 130, flag = 1
id = 4, val = 140, flag = 1
id = 5, val = 50, flag = 0
id = 6, val = 160, flag = 1
id = 7, val = 70, flag = 0
id = 8, val = 80, flag = 0

End

Executed Triggers:

Statement-level BEFORE triggers
Row-level BEFORE triggers:
  for id = 3, 4, 6
Row-level AFTER triggers:
  for id = 3, 4, 6
Statement-level AFTER triggers

Refer to

Transition Tables:

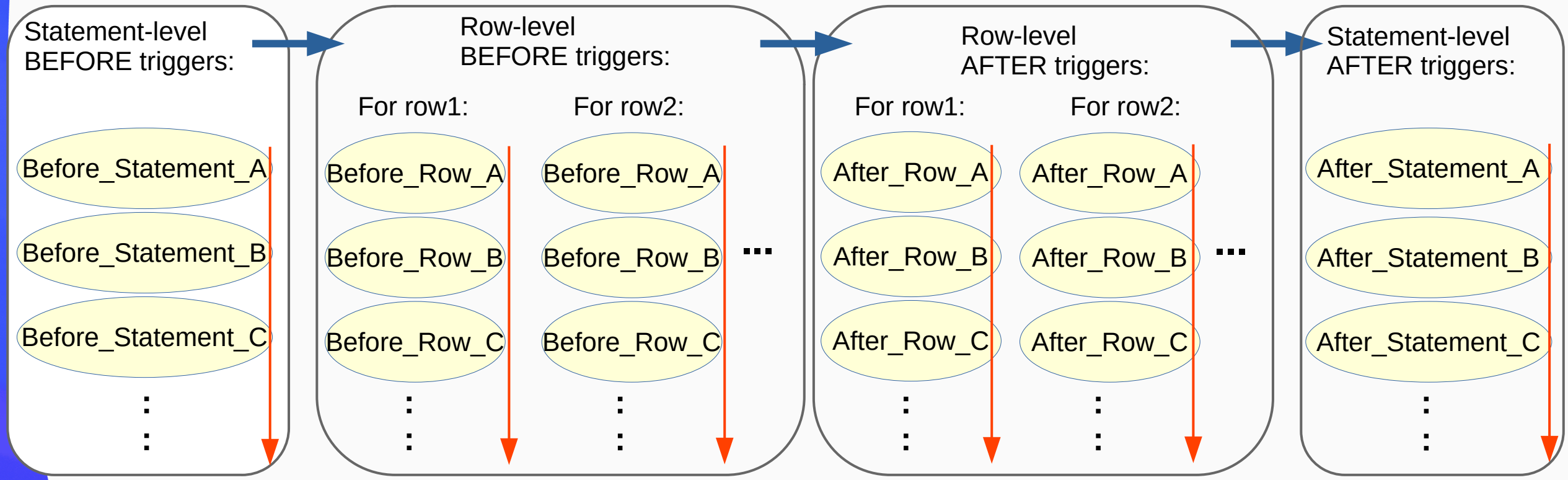
OLD table:
 (id=3, val=30, flag=1)
 (id=4, val=40, flag=1)
 (id=6, val=60, flag=1)

NEW table:
 (id=3, val=130, flag=1)
 (id=3, val=140, flag=1)
 (id=3, val=160, flag=1)

queue:

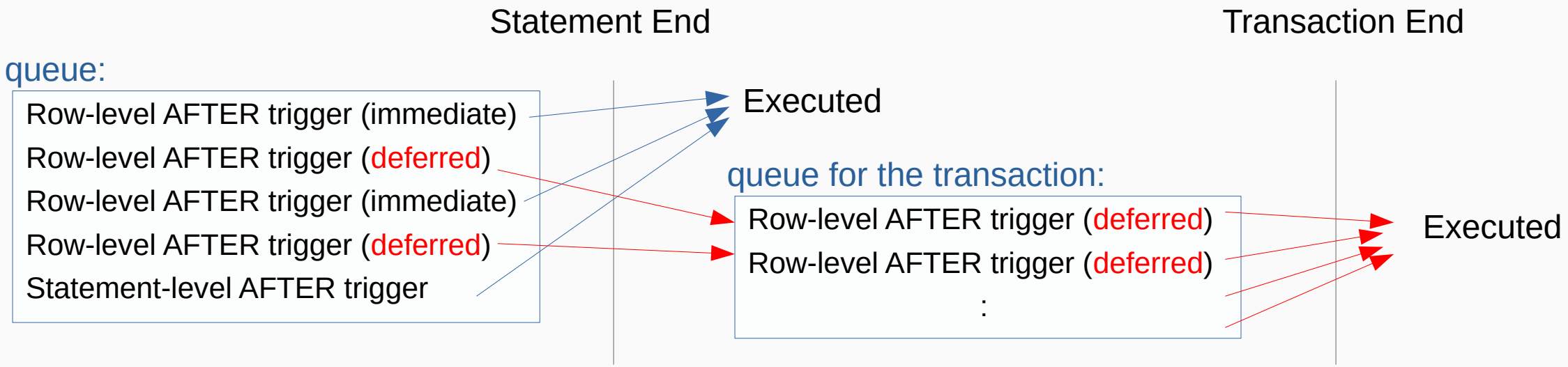
Multiple Triggers on a Table

- Multiple triggers for the same event on the same relation are fired alphabetical order by name.



Deferred Constraint Triggers

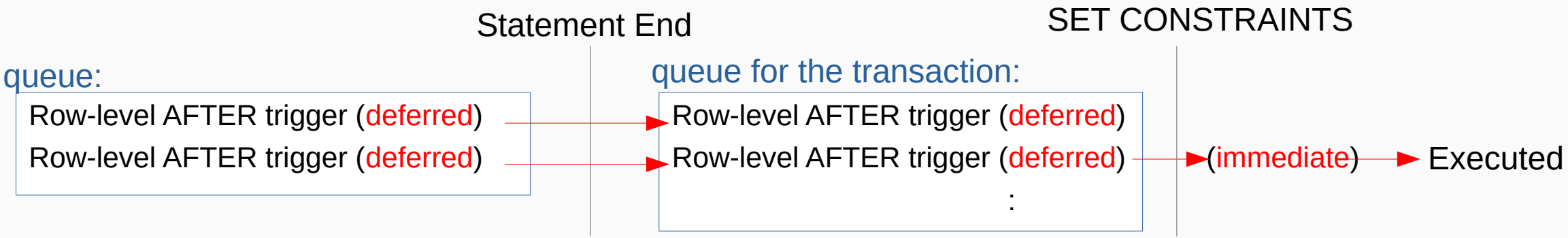
- Not executed at the end of the statement
- Moved to the trigger queue for the transaction
- Executed at the end of the transaction



SET CONSTRAINTS

- Deferred constraint trigger in the queue are executed when changed to *immediate.*

```
BEGIN;

INSERT INTO tbl VALUES(999);
SET CONSTRAINTS tbl_check_constraint IMMEDIATE;
```



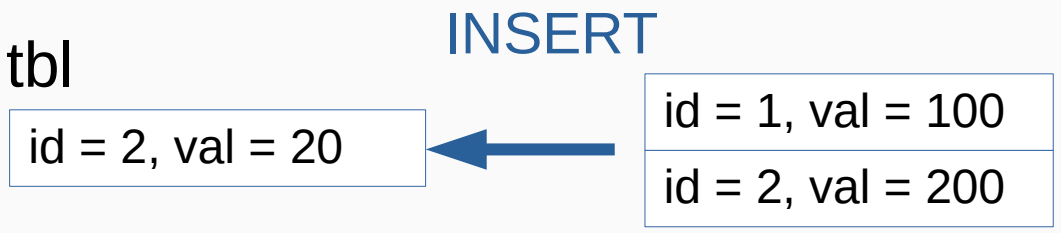
Triggers and Transaction

- Triggers are executed as part of the same transaction as the statement that triggered it.
  - Regardless of whether it is deferred or not
- If either the statement or the trigger causes an error, the effects of both will be rolled back.

Some Complex Situations

INSERT ... ON CONFLICT DO UPDATE

```
INSERT INTO tbl VALUES (1,100), (2,200)
  ON CONFLICT(id) DO UPDATE val = EXCLUDED.val;
```

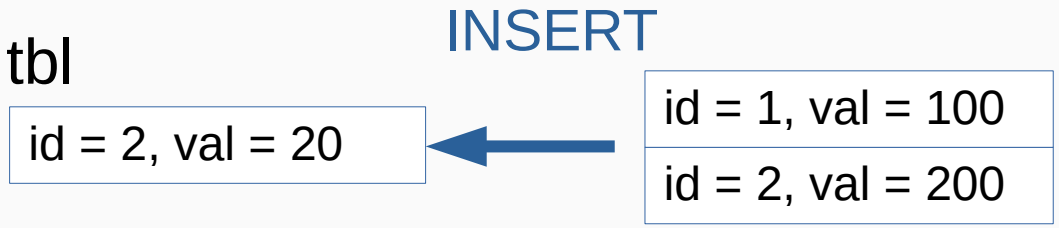


INSERT ... ON CONFLICT DO UPDATE

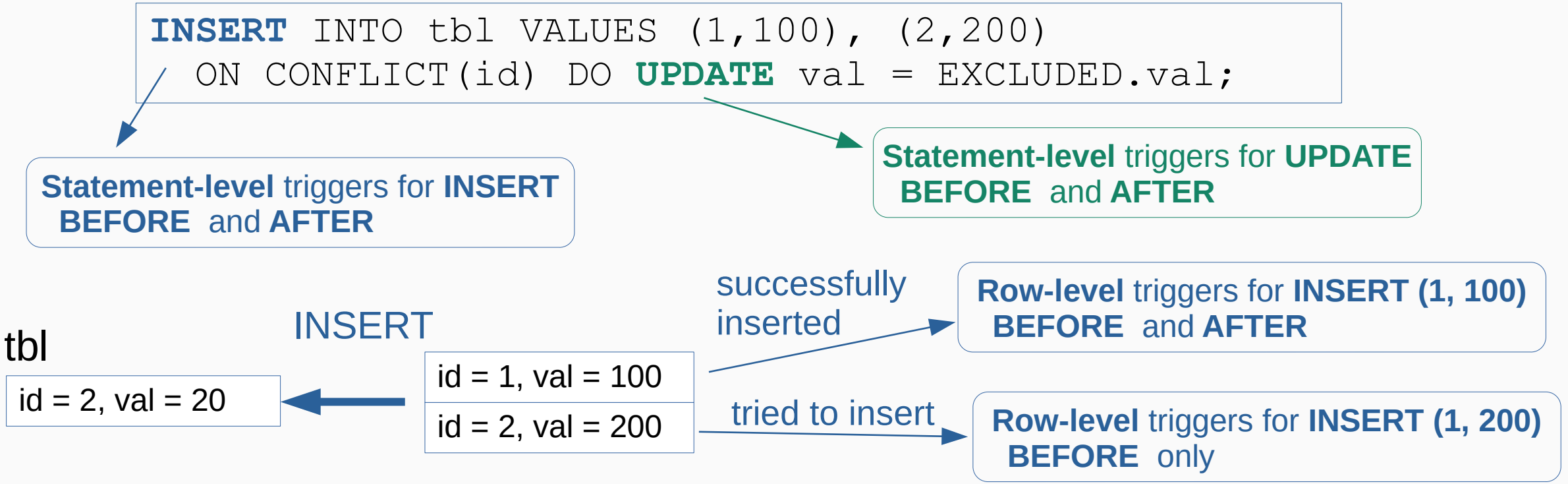
```
INSERT INTO tbl VALUES (1,100), (2,200)
  ON CONFLICT(id) DO UPDATE val = EXCLUDED.val;
```

Statement-level triggers for **INSERT**
BEFORE and **AFTER**

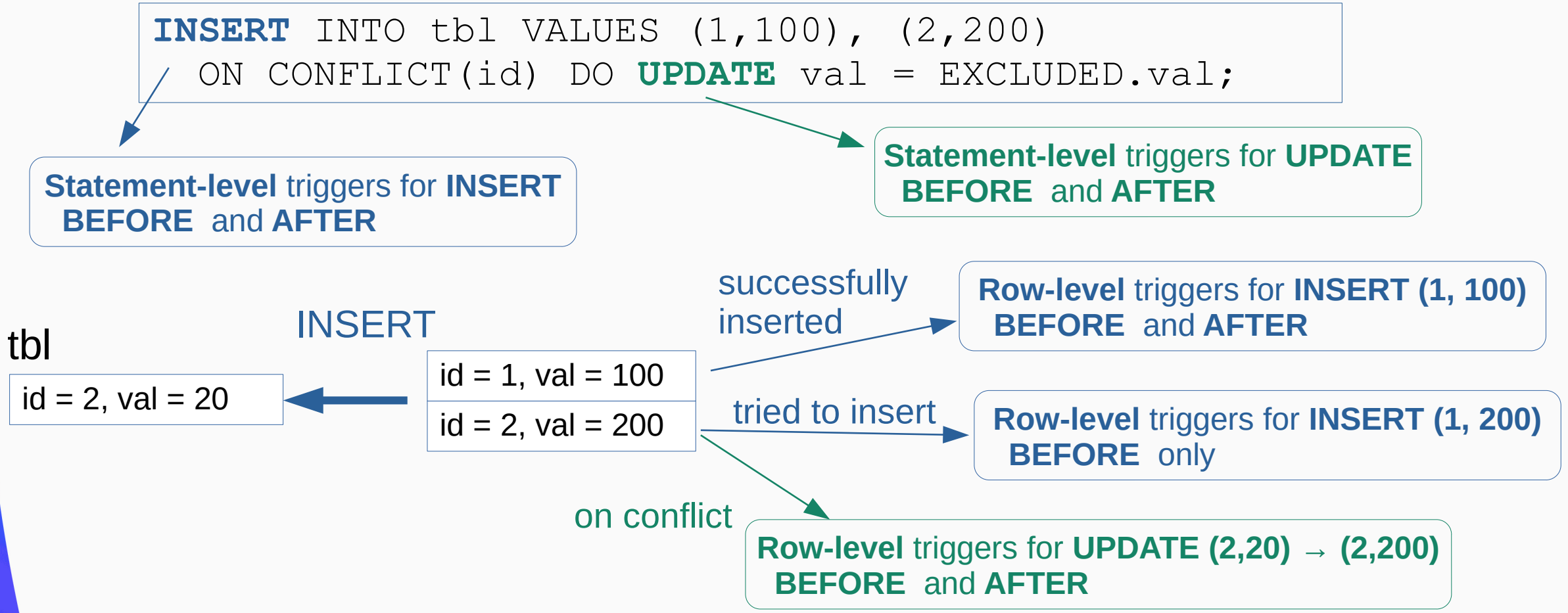
Statement-level triggers for **UPDATE**
BEFORE and **AFTER**



INSERT ... ON CONFLICT DO UPDATE



INSERT ... ON CONFLICT DO UPDATE



MERGE

```
MERGE INTO wines w
USING wine_stock_changes s
ON s.winename = w.winename
WHEN NOT MATCHED AND s.stock_delta > 0 THEN
  INSERT VALUES(s.winename, s.stock_delta)
WHEN MATCHED AND w.stock + s.stock_delta > 0 THEN
  UPDATE SET stock = w.stock + s.stock_delta
WHEN MATCHED THEN
  DELETE;
```

wine_stock_changes
(winename, stock_delta)

‘Chateau Margaux 2000’, 1
'Chateau Lafite 2003', 1
‘Chateau Latour 1997’, -1

MERGE

wines (winename, stock)

'Chateau Lafite 2003', 2
‘Chateau Latour 1997’, 1

MERGE

```
MERGE INTO wines w
USING wine_stock_changes s
ON s.winename = w.winename
WHEN NOT MATCHED AND s.stock_delta > 0 THEN
  INSERT VALUES(s.winename, s.stock_delta)
WHEN MATCHED AND w.stock + s.stock_delta > 0 THEN
  UPDATE SET stock = w.stock + s.stock_delta
WHEN MATCHED THEN
  DELETE;
```

Statement-level triggers for **INSERT**

Statement-level triggers for **UPDATE**

Statement-level triggers for **DELETE**

wine_stock_changes
(winename, stock_delta)

‘Chateau Margaux 2000’, 1
'Chateau Lafite 2003', 1
‘Chateau Latour 1997’, -1

MERGE

wines (winename, stock)

'Chateau Lafite 2003', 2
‘Chateau Latour 1997’, 1

MERGE

```
MERGE INTO wines w
USING wine_stock_changes s
ON s.winename = w.winename
WHEN NOT MATCHED AND s.stock_delta > 0 THEN
  INSERT VALUES(s.winename, s.stock_delta)
WHEN MATCHED AND w.stock + s.stock_delta > 0 THEN
  UPDATE SET stock = w.stock + s.stock_delta
WHEN MATCHED THEN
  DELETE;
```

Statement-level triggers for **INSERT**

Statement-level triggers for **UPDATE**

Statement-level triggers for **DELETE**

wine_stock_changes
(winename, stock_delta)

‘Chateau Margaux 2000’, 1
'Chateau Lafite 2003', 1
‘Chateau Latour 1997’, -1

MERGE

wines (winename, stock)

'Chateau Lafite 2003', 2
‘Chateau Latour 1997’, 1

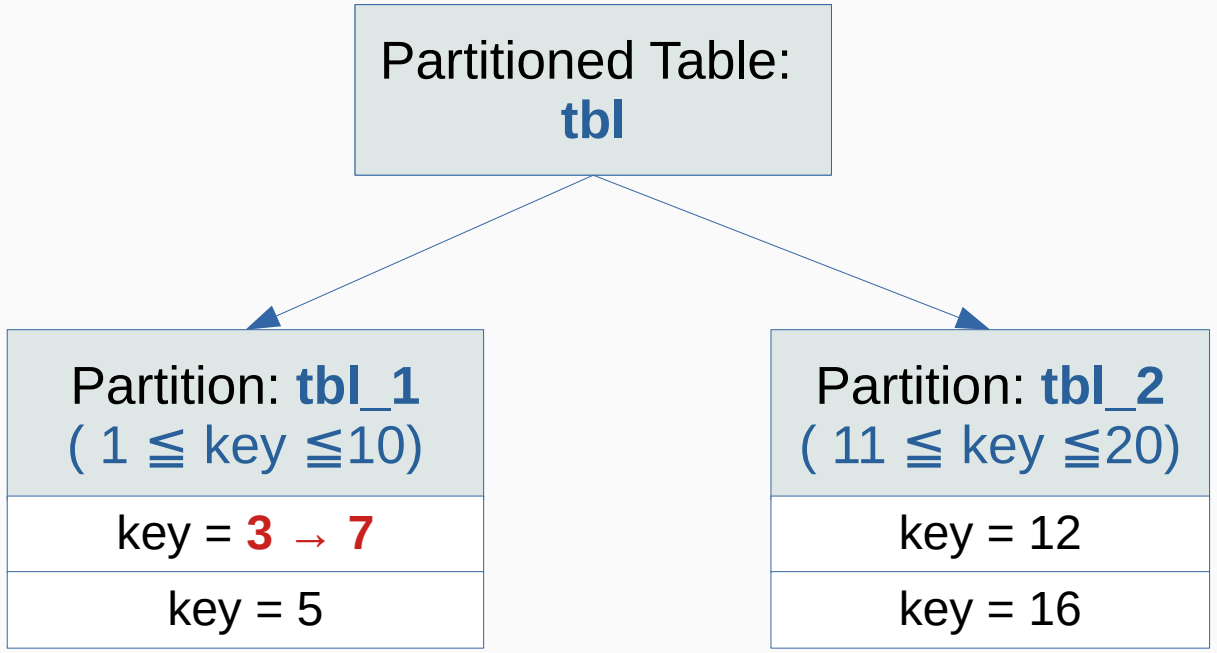
inserted → **Row-level** triggers for **INSERT**

updated → **Row-level** triggers for **UPDATE**

deleted → **Row-level** triggers for **DELETE**

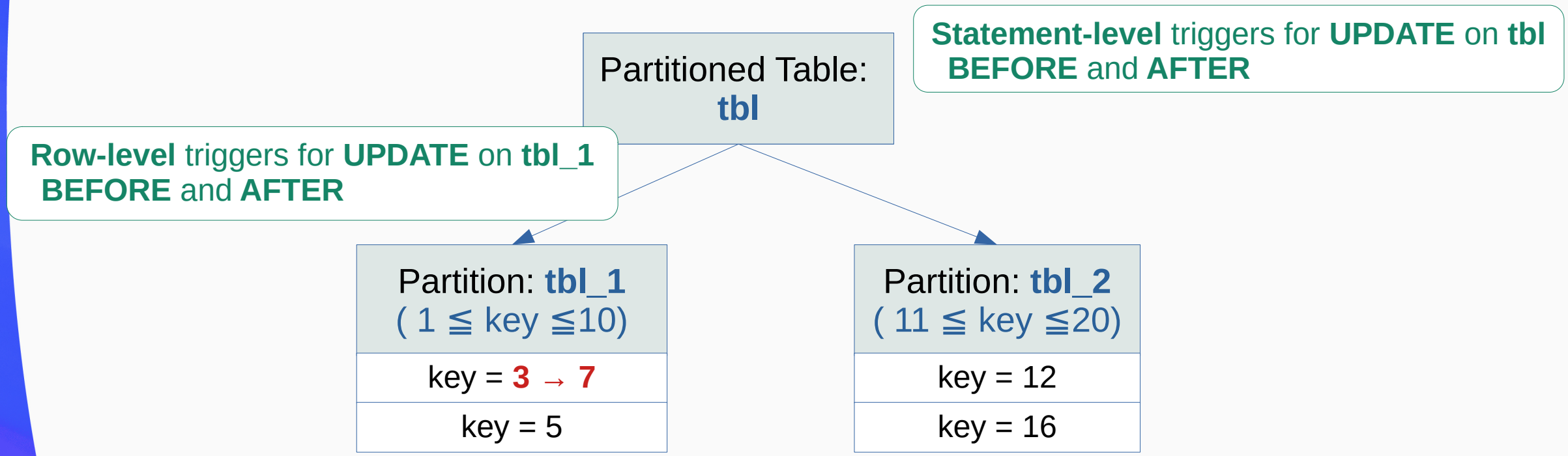
Triggers on Partitioned Table

```
UPDATE tbl SET key = 7 WHERE key = 3;
```



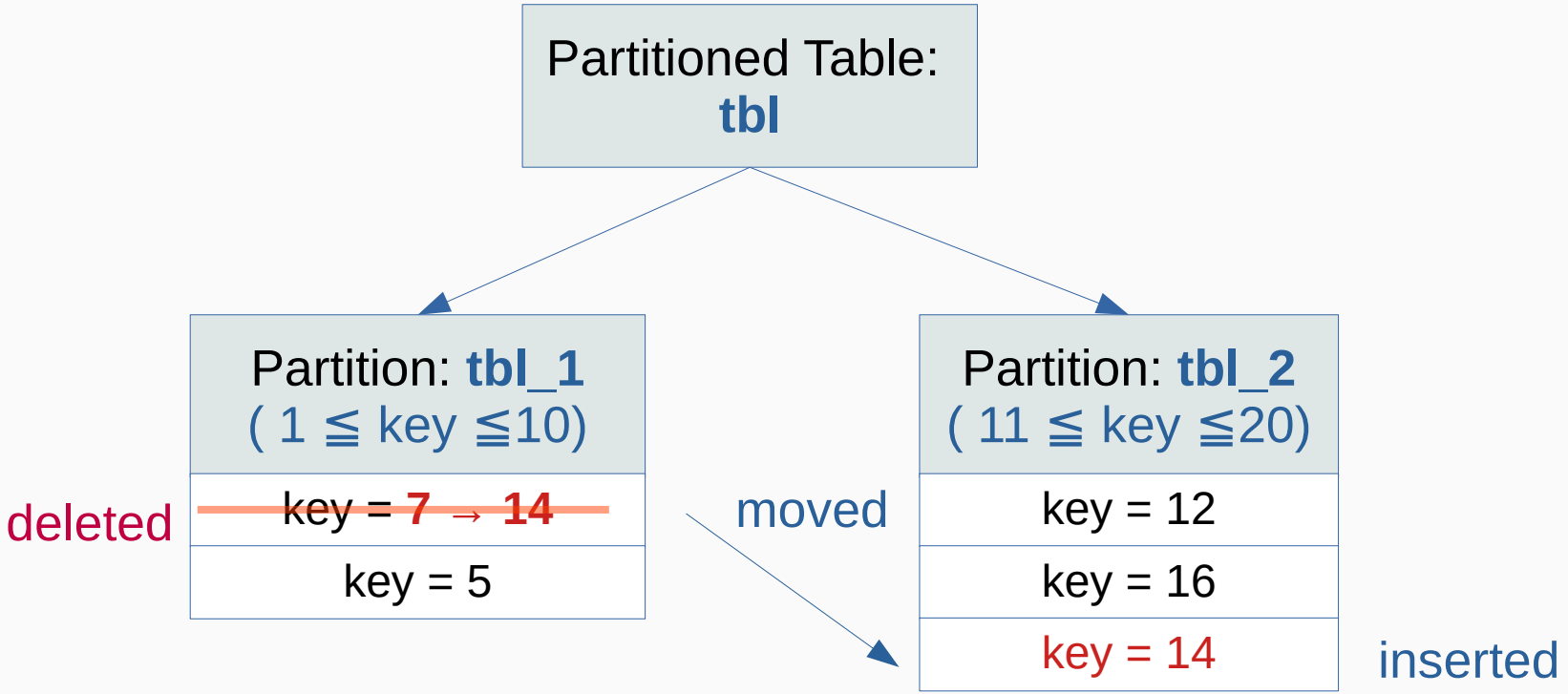
Triggers on Partitioned Table

```
UPDATE tbl SET key = 7 WHERE key = 3;
```



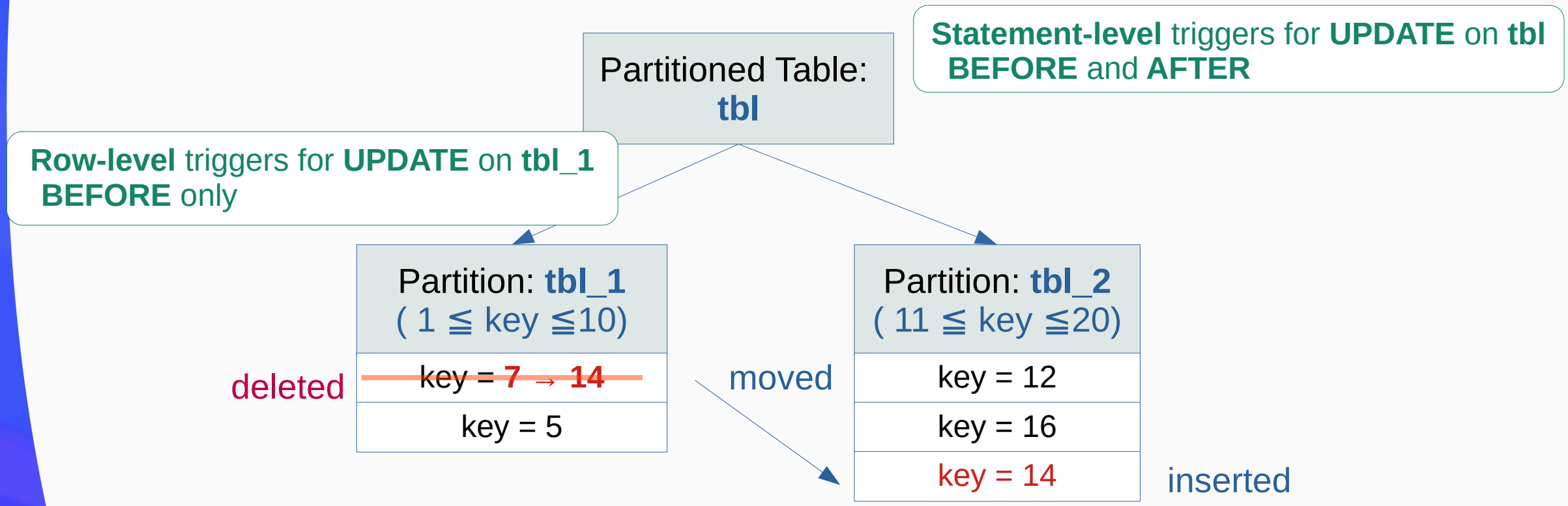
Triggers on Partitioned Table: moving between partitions

```
UPDATE tbl SET key = 14 WHERE key = 7;
```



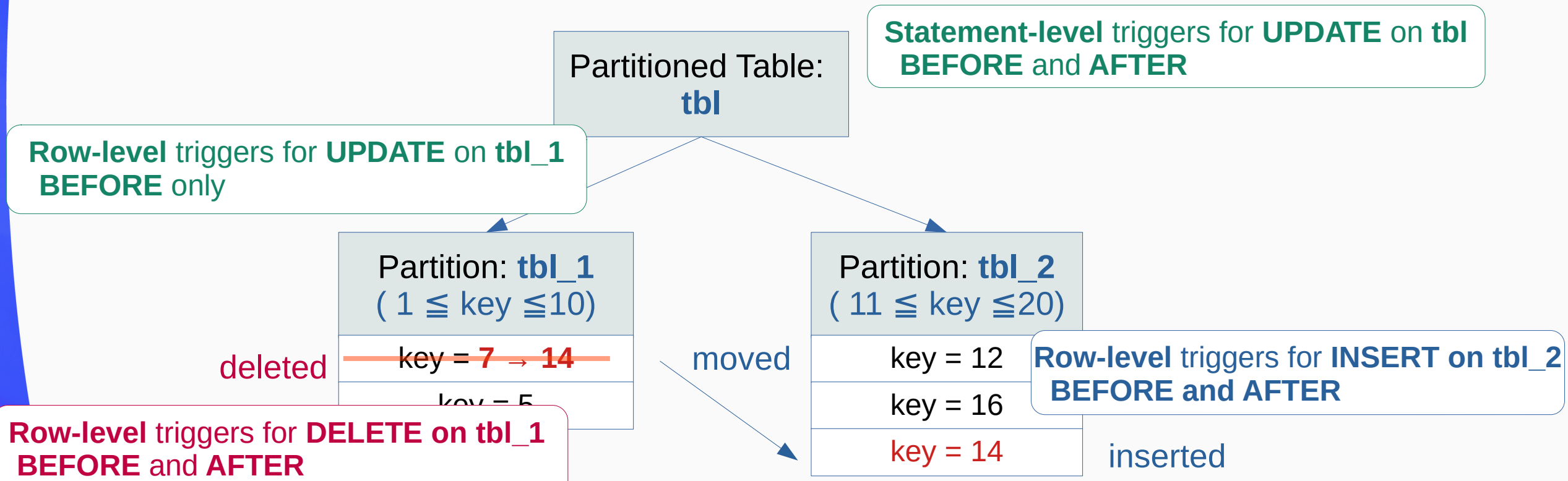
Triggers on Partitioned Table: moving between partitions

```
UPDATE tbl SET key = 14 WHERE key = 7;
```



Triggers on Partitioned Table: moving between partitions

```
UPDATE tbl SET key = 14 WHERE key = 7;
```



Summary

- Triggers
  - Automatically executed whenever a certain type of operation is performed
  - Internally created and used for constraints implementation
- How triggers work
  - How and when it is fired
  - Some complex situations

Thank you!