Авторы: Егор Рогов, Павел Лузанов

Обратная связь
Отзывы, замечания и предложения направляйте по адресу:
edu@postgrespro.ru

Темы

Триггеры и триггерные функции

В какой момент срабатывают триггеры

Контекст выполнения триггерной функции

Возвращаемое значение

Для чего можно применять триггеры и для чего не нужно

Событийные триггеры

Триггеры и функции

Триггер

объект базы данных — список обрабатываемых событий

при возникновении события вызывается триггерная функция и ей передается контекст

Триггерная функция

объект базы данных — код обработки события

выполняется в той же транзакции, что и основная операция

соглашение: функция не принимает параметры, возвращает значение псевдотипа trigger (фактически record)

может использоваться в нескольких триггерах

Механизм триггеров позволяет выполнять определенные действия «в ответ» на определенные события. Триггер состоит из двух частей: собственно триггера (который определяет события) и триггерной функции (которая определяет действия). И триггер, и функция являются самостоятельными объектами БД.

Когда возникает событие, на которое «подписан» триггер, вызывается триггерная функция. Ей передается контекст вызова, чтобы можно было определить, какой именно триггер и в каких условиях вызвал функцию.

Триггерная функция — это обычная функция, которая написана с учетом некоторых соглашений:

- она пишется на любом языке, кроме чистого SQL;

- она не имеет параметров;

- она возвращает значение типа trigger (на самом деле это псевдотип, по факту возвращается запись, соответствующая строке таблицы).

Триггерная функция выполняется в той же транзакции, что и основная операция. Таким образом, если триггерная функция завершится с ошибкой, вся транзакция будет прервана.

https://postgrespro.ru/docs/postgresql/12/trigger-definition

События

INSERT, UPDATE, DELETE

таблицы	before/after	statement
	before/after	row
представления	before/after	statement
	instead of	row

TRUNCATE

таблицы	before/after	statement

Условие WHEN

устанавливает дополнительный фильтр

Триггеры могут срабатывать на вставку (INSERT), обновление (UPDATE) или удаление (DELETE) строк в таблице или представлении, а также на опустошение (TRUNCATE) таблиц.

Триггер может срабатывать до выполнения действия (BEFORE), после него (AFTER), или вместо него (INSTEAD OF).

Триггер может срабатывать один раз для всей операции (FOR EACH STATEMENT), или отдельно для каждой затронутой строки (FOR EACH ROW).

Не для любой комбинации этих условий можно создать триггер. Например, instead-of-триггеры можно определить только для представлений на уровне строк, а truncate-триггер можно определить только для таблиц и только на уровне оператора. Допустимые варианты перечислены на слайде.

Кроме того, можно сузить область действия триггера, указав дополнительное условие WHEN: если условие не выполняется — триггер не срабатывает.

https://postgrespro.ru/docs/postgresql/12/sql-createtrigger

Before statement

Срабатывает

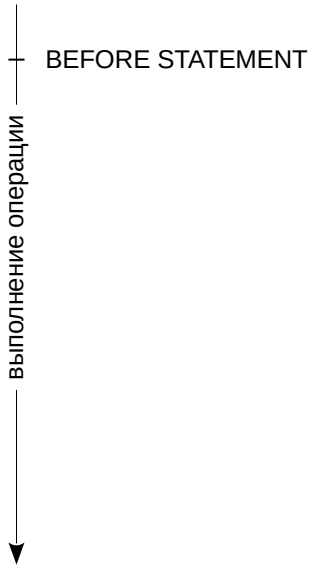
до операции

Возвращаемое значение

игнорируется

Контекст

TG-переменные



Рассмотрим подробнее разные типы триггеров.

Триггер BEFORE STATEMENT срабатывает один раз для операции независимо от того, сколько строк будет затронуто (возможно, что и не одной). Это происходит до того, как операция начала выполняться.

Возвращаемое значение триггерной функции игнорируется, можно возвращать просто NULL. Если в триггере возникает ошибка, операция отменяется.

Поскольку триггерная функция не имеет параметров, контекст вызова в PL/pgSQL передается ей с помощью предопределенных TG-переменных, таких, как:

- TG_WHEN = «BEFORE»,

- TG_LEVEL = «STATEMENT»,

- TG_OP = «INSERT»/«UPDATE»/«DELETE»/«TRUNCATE»

и др. Триггерной функции можно также передать пользовательский контекст (аналог отсутствующих параметров) через переменную TG_ARGV, хотя зачастую лучше вместо одной «обобщенной» функции создать несколько «частных».

https://postgrespro.ru/docs/postgresql/12/plpgsql-trigger

Before row

Срабатывает

перед действием со строкой
в процессе выполнения операции

Возвращаемое значение

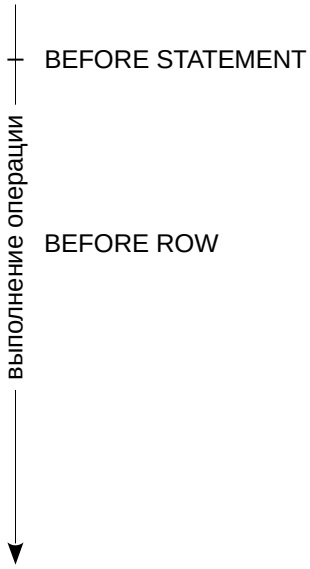
строка (возможно, измененная)

null отменяет действие

Контекст

OLD		update, delete
NEW	insert,	update

TG-переменные



Триггеры BEFORE ROW срабатывают каждый раз перед тем, как операция затронет строку. Это происходит непосредственно в процессе выполнения операции.

В качестве контекста триггерная функция получает переменные:

- OLD — исходное состояние строки (не определено для вставки),
- NEW — измененное состояние строки (не определено для удаления),
- TG_WHEN = «BEFORE»,
- TG_LEVEL = «ROW»,
- TG_OP = «INSERT»/«UPDATE»/«DELETE»

и др.

Возврат неопределенного значения NULL воспринимается как отмена действия над данной строкой. Сама операция продолжит выполнение, но текущая строка не будет обработана, и другие триггеры для этой строки не сработают.

Чтобы не вмешиваться в работу операции, триггер должен вернуть строку ровно в том виде, в котором он ее получил. Для этого при вставке или обновлении нужно вернуть NEW, а при удалении — любое значение кроме NULL (обычно используют OLD).

Но триггерная функция может и изменить значение NEW, чтобы повлиять на результат операции — часто именно для этого такой триггер и создают.

Instead of row

Срабатывает

вместо действия со строкой
для представлений

Возвращаемое значение

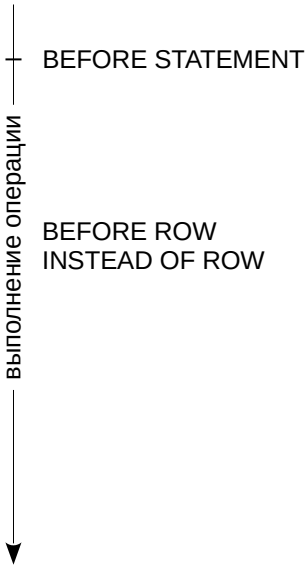
строка (возможно, измененная) —
будет видна в RETURNING

null отменяет действие

Контекст

OLD update, delete
NEW insert, update

TG-переменные



Триггеры INSTEAD OF очень похожи на триггеры BEFORE, но определяются только для представлений и срабатывают не до, а вместо операции.

В задачу таких триггеров обычно входит выполнение необходимых операций над базовыми таблицами представления. Также триггер может вернуть измененное значение NEW — именно оно будет видно при выполнении операции с указанием фразы RETURNING.

After row

Срабатывает

после выполнения операции

очередь из прошедших условие WHEN

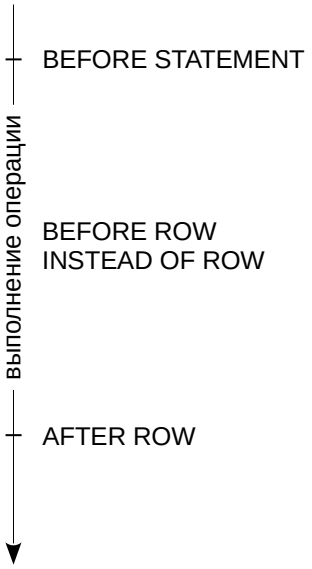
Возвращаемое значение

игнорируется

Контекст

OLD, OLD TABLE		update, delete
NEW, NEW TABLE	insert,	update

TG-переменные



Триггеры AFTER ROW, как и BEFORE ROW, срабатывают для каждой затрагиваемой строки, но не сразу после действия над строкой, а после того, как выполнена вся операция. Для этого события сначала помещаются в очередь и обрабатываются после окончания операции. Чем меньше событий попадет в очередь, тем меньше будет накладных расходов — поэтому именно в этом случае очень полезно использовать условие WHEN, чтобы отсечь заведомо ненужные строки.

Возвращаемое значение триггеров AFTER ROW игнорируется (поскольку операция уже выполнена).

Контекст триггерной функции составляют:

- OLD — исходное состояние строки (не определено для вставки),

- NEW — новое состояние строки (не определено для удаления).

Кроме этих переменных, начиная с версии 10, триггерная функция может получить доступ к специальным *переходным таблицам* (transition tables). Таблица, указанная при создании триггера как OLD TABLE, содержит старые значения строк, обработанных триггером, а таблица NEW TABLE — новые значения тех же строк.

Доступны и обычные TG-переменные, включая:

- TG_WHEN = «AFTER»,

- TG_LEVEL = «ROW»,

- TG_OP = «INSERT»/«UPDATE»/«DELETE».

After statement

Срабатывает

после операции
(даже если не затронута ни одна строка)

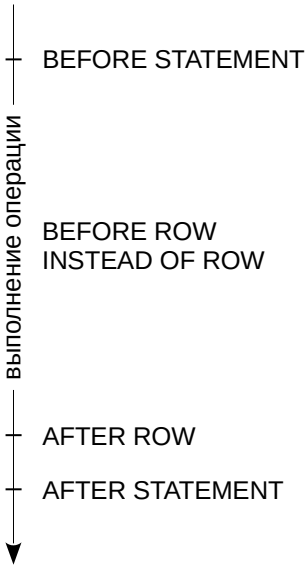
Возвращаемое значение

игнорируется

Контекст

OLD TABLE		update, delete
NEW TABLE	insert,	update

TG-переменные



Триггер AFTER STATEMENT срабатывает после того, как операция завершится и отработают все триггеры AFTER ROW. Этот триггер срабатывает ровно один раз независимо от того, сколько строк было затронуто операцией.

Возвращаемое значение триггерной функции игнорируется.

Контекст вызова передается с помощью переходных таблиц. Обращаясь к ним, триггерная функция может проанализировать все затронутые строки. Обычно переходные таблицы используются именно с триггерами AFTER STATEMENT, а не AFTER ROW.

Также определены обычные TG-переменные:

- TG_WHEN = «AFTER»,

- TG_LEVEL = «STATEMENT»,

- TG_OP = «INSERT»/«UPDATE»/«DELETE»/«TRUNCATE»

и др.

Порядок вызова триггеров

Создадим «универсальную» триггерную функцию, которая описывает контекст, в котором она вызвана. Контекст передается в различных TG-переменных.

Затем создадим триггеры на различные события и будем смотреть, какие триггеры вызываются при выполнении операций и в каком порядке.

```
=> CREATE OR REPLACE FUNCTION describe() RETURNS trigger
AS $$
DECLARE
    rec record;
    str text := '';
BEGIN
    IF TG_LEVEL = 'ROW' THEN
        CASE TG_OP
            WHEN 'DELETE' THEN rec := OLD; str := OLD::text;
            WHEN 'UPDATE' THEN rec := NEW; str := OLD || ' -> ' || NEW;
            WHEN 'INSERT' THEN rec := NEW; str := NEW::text;
        END CASE;
    END IF;
    RAISE NOTICE '% % % %: %',
        TG_TABLE_NAME, TG_WHEN, TG_OP, TG_LEVEL, str;
    RETURN rec;
END;
$$ LANGUAGE plpgsql;
```

```
CREATE FUNCTION
```

Таблица:

```
=> CREATE TABLE t(
    id integer PRIMARY KEY,
    s text
);
```

```
CREATE TABLE
```

Триггеры на уровне оператора.

```
=> CREATE TRIGGER t_before_stmt
BEFORE INSERT OR UPDATE OR DELETE -- события
ON t                              -- таблица
FOR EACH STATEMENT                -- уровень
EXECUTE FUNCTION describe();      -- триггерная функция
```

```
CREATE TRIGGER
```

```
=> CREATE TRIGGER t_after_stmt
AFTER INSERT OR UPDATE OR DELETE ON t
FOR EACH STATEMENT EXECUTE FUNCTION describe();
```

```
CREATE TRIGGER
```

И на уровне строк:

```
=> CREATE TRIGGER t_before_row
BEFORE INSERT OR UPDATE OR DELETE ON t
FOR EACH ROW EXECUTE FUNCTION describe();
```

```
CREATE TRIGGER
```

```
=> CREATE TRIGGER t_after_row
AFTER INSERT OR UPDATE OR DELETE ON t
FOR EACH ROW EXECUTE FUNCTION describe();
```

```
CREATE TRIGGER
```

Пробуем вставку:

```
=> INSERT INTO t VALUES (1,'aaa');
```

```
NOTICE:  t BEFORE INSERT STATEMENT:
NOTICE:  t BEFORE INSERT ROW: (1,aaa)
NOTICE:  t AFTER INSERT ROW: (1,aaa)
NOTICE:  t AFTER INSERT STATEMENT:
INSERT 0 1
```

Обновление:

```
=> UPDATE t SET s = 'bbb';
```

```
NOTICE:  t BEFORE UPDATE STATEMENT:
NOTICE:  t BEFORE UPDATE ROW: (1,aaa) -> (1,bbb)
NOTICE:  t AFTER UPDATE ROW: (1,aaa) -> (1,bbb)
NOTICE:  t AFTER UPDATE STATEMENT:
UPDATE 1
```

Триггеры на уровне оператора сработают даже если команда не обработала ни одной строки:

```
=> UPDATE t SET s = 'ccc' where id = 0;
```

```
NOTICE:  t BEFORE UPDATE STATEMENT:
NOTICE:  t AFTER UPDATE STATEMENT:
UPDATE 0
```

Тонкий момент: оператор INSERT с предложением ON CONFLICT приводит к тому, что срабатывают BEFORE-триггеры и на вставку, и на обновление:

```
=> INSERT INTO t VALUES (1,'ccc'), (3,'ddd')
ON CONFLICT(id) DO UPDATE SET s = EXCLUDED.s;
```

```
NOTICE:  t BEFORE INSERT STATEMENT:
NOTICE:  t BEFORE UPDATE STATEMENT:
NOTICE:  t BEFORE INSERT ROW: (1,ccc)
NOTICE:  t BEFORE UPDATE ROW: (1,bbb) -> (1,ccc)
NOTICE:  t BEFORE INSERT ROW: (3,ddd)
NOTICE:  t AFTER UPDATE ROW: (1,bbb) -> (1,ccc)
NOTICE:  t AFTER INSERT ROW: (3,ddd)
NOTICE:  t AFTER UPDATE STATEMENT:
NOTICE:  t AFTER INSERT STATEMENT:
INSERT 0 2
```

И, наконец, удаление:

```
=> DELETE FROM t;
```

```
NOTICE:  t BEFORE DELETE STATEMENT:
NOTICE:  t BEFORE DELETE ROW: (1,ccc)
NOTICE:  t BEFORE DELETE ROW: (3,ddd)
NOTICE:  t AFTER DELETE ROW: (1,ccc)
NOTICE:  t AFTER DELETE ROW: (3,ddd)
NOTICE:  t AFTER DELETE STATEMENT:
DELETE 2
```

Переходные таблицы

Напишем триггерную функцию, показывающую содержимое переходных таблиц. Здесь мы используем имена old_table и new_table, которые будут объявлены при создании триггера.

Переходные таблицы «выглядят» настоящими, но не присутствуют в системном каталоге и располагаются в оперативной памяти (хотя и могут сбрасываться на диск при большом объеме).

```
=> CREATE OR REPLACE FUNCTION transition() RETURNS trigger
AS $$
DECLARE
    rec record;
BEGIN
    IF TG_OP = 'DELETE' OR TG_OP = 'UPDATE' THEN
        RAISE NOTICE 'Старое состояние:';
        FOR rec IN SELECT * FROM old_table LOOP
            RAISE NOTICE '%', rec;
        END LOOP;
    END IF;
    IF TG_OP = 'UPDATE' OR TG_OP = 'INSERT' THEN
        RAISE NOTICE 'Новое состояние:';
        FOR rec IN SELECT * FROM new_table LOOP
            RAISE NOTICE '%', rec;
        END LOOP;
    END IF;
    RETURN NULL;
END;
$$ LANGUAGE plpgsql;
```

```
CREATE FUNCTION
```

Создадим новую таблицу:

```
=> CREATE TABLE trans(

    id integer PRIMARY KEY,
    n integer
);
```

```
CREATE TABLE
```

```
=> INSERT INTO trans VALUES (1,10), (2,20), (3,30);
```

```
INSERT 0 3
```

Чтобы при выполнении операции создавались переходные таблицы, необходимо указать их имена при создании триггера:

```
=> CREATE TRIGGER t_after_upd_trans
AFTER UPDATE ON trans -- только одно событие на один триггер
REFERENCING
    OLD TABLE AS old_table
    NEW TABLE AS new_table -- можно и одну, не обязательно обе
FOR EACH STATEMENT
EXECUTE FUNCTION transition();
```

```
CREATE TRIGGER
```

Проверим:

```
=> UPDATE trans SET n = n + 1 WHERE n <= 20;
```

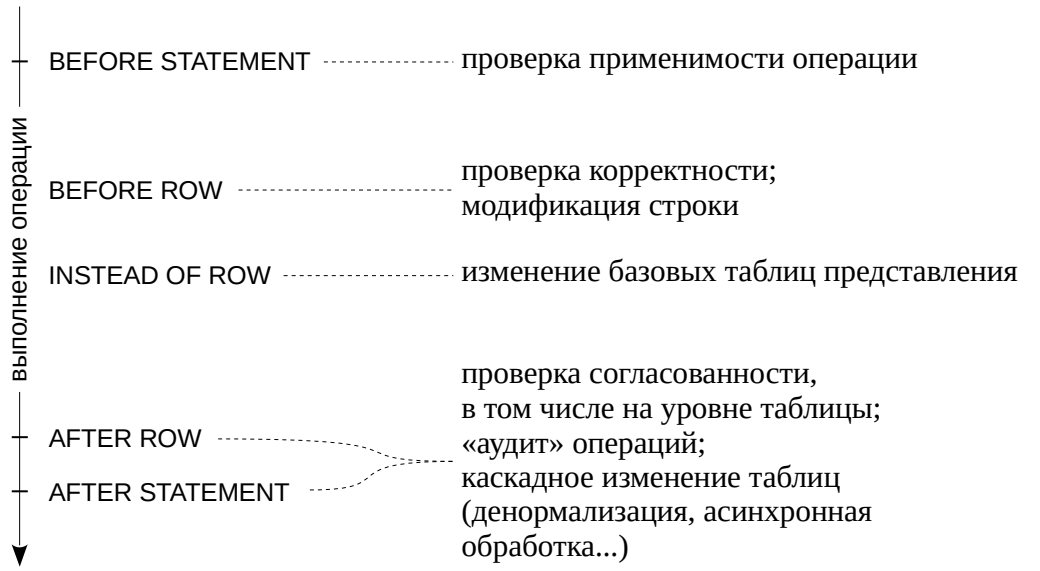
```
NOTICE:  Старое состояние:
NOTICE:  (1,10)
NOTICE:  (2,20)
NOTICE:  Новое состояние:
NOTICE:  (1,11)
NOTICE:  (2,21)
UPDATE 2
```

Переходные таблицы содержат только те строки, которые были затронуты операцией.

Для вставки и удаления переходные таблицы работают точно так же, как и для обновления, но доступны только NEW TABLE и OLD TABLE соответственно.

Поскольку триггеры AFTER ROW срабатывают после выполнения всей операции, переходные таблицы можно использовать и в них. Но обычно это не имеет смысла.

Возможное использование



Каково практическое применение триггеров?

BEFORE-триггеры можно использовать для проверки корректность операции и при необходимости вызвать ошибку.

Триггеры BEFORE ROW можно применять для модификации строки (например, заполнить пустое поле нужным значением). Это бывает удобно, чтобы не повторять логику заполнения «технических» полей в каждой операции, а также позволяет вмешаться в работу приложения, код которого недоступен для изменения.

Триггеры INSTEAD OF ROW применяют для того, чтобы отобразить операции над представлением в операции над базовыми таблицами.

Триггеры AFTER ROW и AFTER STATEMENT полезны в случаях, когда нужно знать точное состояние после операции (BEFORE-триггеры могут влиять на результат, так что на этом этапе ясности еще нет):

- для проверки согласованности операции;

- для «аудита», то есть записи изменений в отдельное хранилище;

- для каскадного изменения других таблиц (например, обновлять денормализованные данные при изменении базовых таблиц, или записывать изменения в очередь для последующей обработки вне данной транзакции).

Если операции затрагивают сразу много строк, то триггер AFTER STATEMENT с переходными таблицами может оказаться более эффективным решением, чем триггер AFTER ROW, поскольку позволяет обрабатывать все изменения пакетно, а не построчно.

Сложности

Код вызывается неявно

- сложно отследить логику выполнения

Правила видимости изменчивой триггерной функции

- виден результат триггеров BEFORE ROW или INSTEAD OF ROW

Порядок вызова триггеров для одного события

- триггеры отрабатывают в алфавитном порядке

Не предотвращается зацикливание

- триггер может вызвать срабатывание других триггеров

Можно нарушить ограничения целостности

- например, исключив из обработки строки, которые должны удалиться

Однако не стоит злоупотреблять триггерами. Триггеры срабатывают неявно, что сильно затрудняет понимание логики работы и крайне усложняет поддержку приложения. Попытки реализовать сложную логику на триггерах обычно заканчиваются плачевно.

В некоторых случаях вместо триггеров можно использовать вычисляемые поля (GENERATED ALWAYS AS … STORED). Такое решение — если оно подходит — будет заведомо проще и прозрачнее.

Есть ряд тонкостей, связанных с триггерами, которые мы сознательно не рассматриваем подробно:

- правила видимости изменчивых (volatile) функций в триггерах BEFORE ROW и INSTEAD OF ROW (не стоит обращаться к таблице, полагаясь на порядок, в котором сработают триггеры);

- порядок вызова нескольких триггеров, обрабатывающих одно и то же событие (не стоит усугублять и без того неявное срабатывание триггеров завязкой на последовательность обработки);

- возможность зацикливания в случае каскадного срабатывания других триггеров, которые, в свою очередь, могут приводить с новым срабатываниям данного триггера;

- возможность нарушить ограничения целостности (например, при исключении из обработки строки, которая удаляется условием ON DELETE CASCADE, может быть нарушена ссылочная целостность).

Если вы столкнулись с тем, что эти тонкости важны для вашего приложения — серьезно задумайтесь.

Примеры использования триггеров

Пример 1: сохранение истории изменения строк.

Пусть есть таблица, содержащая актуальные данные. Задача состоит в том, чтобы в отдельной таблице сохранять всю историю изменения строк основной таблицы.

Поддержку исторической таблицы можно было бы возложить на приложение, но тогда велика вероятность, что в каких-то случаях из-за ошибок история не будет сохраняться. Поэтому решим задачу с помощью триггеров.

Основная таблица:

```
=> CREATE TABLE coins(
    face_value numeric,
    name text
);
```

```
CREATE TABLE
```

Мы рассчитываем, что название исторической таблицы образуется от названия основной добавлением «_history». Сначала создаем клон основной таблицы...

```
=> CREATE TABLE coins_history(LIKE coins);
```

```
CREATE TABLE
```

...и затем добавляем столбцы «действительно с» и «действительно по»:

```
=> ALTER TABLE coins_history
    ADD start_date timestamp,
    ADD end_date timestamp;
```

```
ALTER TABLE
```

Одна триггерная функция будет вставлять новую историческую строку с открытым интервалом действия:

```
=> CREATE OR REPLACE FUNCTION history_insert() RETURNS trigger
AS $$
BEGIN
    EXECUTE format(
        'INSERT INTO %I SELECT ($1).*, current_timestamp, NULL',
        TG_TABLE_NAME||'_history'
    ) USING NEW;

    RETURN NEW;
END;
$$ LANGUAGE plpgsql;
```

```
CREATE FUNCTION
```

Другая функция будет закрывать интервал действия исторической строки:

```
=> CREATE OR REPLACE FUNCTION history_delete() RETURNS trigger
AS $$
BEGIN
    EXECUTE format(
        'UPDATE %I SET end_date = current_timestamp WHERE face_value = $1 AND end_date IS NULL',
        TG_TABLE_NAME||'_history'
    ) USING OLD.face_value;

    RETURN OLD;
END;
$$ LANGUAGE plpgsql;
```

```
CREATE FUNCTION
```

Теперь создадим триггеры. Важные моменты:

- Обновление трактуется как удаление и затем вставка; здесь важен порядок, в котором сработают триггеры (по алфавиту).
- Current_timestamp возвращает время начала транзакции, поэтому при обновлении start_date одной строки будет равен end_date другой.
- Использование AFTER-триггеров позволяет избежать проблем с INSERT ... ON CONFLICT и потенциальными конфликтами с другими триггерами, которые могут существовать на основной таблице.

```
=> CREATE TRIGGER coins_history_insert
AFTER INSERT OR UPDATE ON coins
FOR EACH ROW EXECUTE FUNCTION history_insert();
```

```
CREATE TRIGGER
```

```
=> CREATE TRIGGER coins_history_delete
AFTER UPDATE OR DELETE ON coins
FOR EACH ROW EXECUTE FUNCTION history_delete();
```

```
CREATE TRIGGER
```

Проверим работу триггеров.

```
=> INSERT INTO coins VALUES (0.25, 'Полушка'), (3, 'Алтын');
```

```
INSERT 0 2
```

```
=> UPDATE coins SET name = '3 копейки' WHERE face_value = 3;
```

```
UPDATE 1
```

```
=> INSERT INTO coins VALUES (5, '5 копеек');
```

```
INSERT 0 1
```

```
=> DELETE FROM coins WHERE face_value = 0.25;
```

```
DELETE 1
```

```
=> SELECT * FROM coins;
```

```
 face_value |   name
------------+-----------
          3 | 3 копейки
          5 | 5 копеек
(2 rows)
```

В исторической таблице хранится вся история изменений:

```
=> SELECT * FROM coins_history ORDER BY face_value, start_date;
```

```
 face_value |   name    |         start_date         |          end_date
------------+-----------+----------------------------+----------------------------
       0.25 | Полушка   | 2022-11-17 11:18:38.703811 | 2022-11-17 11:18:43.919679
          3 | Алтын     | 2022-11-17 11:18:38.703811 | 2022-11-17 11:18:40.809421
          3 | 3 копейки | 2022-11-17 11:18:40.809421 |
          5 | 5 копеек  | 2022-11-17 11:18:41.87551  |
(4 rows)
```

И теперь по ней можно восстановить состояние на любой момент времени (это напоминает работу механизма MVCC). Например, на самое начало:

```
=> \set d '2022-11-17 11:18:38.760153+03'
```

```
=> SELECT face_value, name
FROM coins_history
WHERE start_date <= :'d' AND (end_date IS NULL OR :'d' < end_date)
ORDER BY face_value;
```

```
 face_value |  name
------------+---------
       0.25 | Полушка
          3 | Алтын
(2 rows)
```

Примеры использования триггеров

Пример 2: обновляемое представление.

Пусть имеются две таблицы: аэропорты и рейсы:

```
=> CREATE TABLE airports(
    code char(3) PRIMARY KEY,
    name text NOT NULL
);
```

```
CREATE TABLE
```

```
=> INSERT INTO airports VALUES
    ('SVO', 'Москва. Шереметьево'),
    ('LED', 'Санкт-Петербург. Пулково'),
    ('TOF', 'Томск. Богашево');
```

```
INSERT 0 3
```

```
=> CREATE TABLE flights(
    id integer PRIMARY KEY GENERATED ALWAYS AS IDENTITY,
    airport_from char(3) NOT NULL REFERENCES airports(code),
    airport_to   char(3) NOT NULL REFERENCES airports(code),
    UNIQUE (airport_from, airport_to)
);

CREATE TABLE

=> INSERT INTO flights(airport_from, airport_to) VALUES
    ('SVO','LED');

INSERT 0 1
```

Для удобства можно определить представление:

```
=> CREATE VIEW flights_v AS
SELECT id,
       (SELECT name
        FROM airports
        WHERE code = airport_from) airport_from,
       (SELECT name
        FROM airports
        WHERE code = airport_to) airport_to
FROM flights;

CREATE VIEW

=> SELECT * FROM flights_v;

 id |    airport_from     |        airport_to
----+---------------------+--------------------------
  1 | Москва. Шереметьево | Санкт-Петербург. Пулково
(1 row)
```

Но такое представление не допускает изменений. Например, не получится изменить пункт назначения таким образом:

```
=> UPDATE flights_v
SET airport_to = 'Томск. Богашево'
WHERE id = 1;

ERROR:  cannot update column "airport_to" of view "flights_v"
DETAIL:  View columns that are not columns of their base relation are not updatable.
```

Однако мы можем определить триггер. Триггерная функция может выглядеть, например, так (для краткости обрабатываем только аэропорт назначения, но не составит труда добавить и аэропорт вылета):

```
=> CREATE OR REPLACE FUNCTION flights_v_update() RETURNS trigger
AS $$
DECLARE
    code_to char(3);
BEGIN
    BEGIN
        SELECT code INTO STRICT code_to
        FROM airports
        WHERE name = NEW.airport_to;
    EXCEPTION
        WHEN no_data_found THEN
            RAISE EXCEPTION 'Аэропорт % отсутствует', NEW.airport_to;
    END;
    UPDATE flights
    SET airport_to = code_to
    WHERE id = OLD.id; -- изменение id игнорируем
    RETURN NEW;
END;
$$ LANGUAGE plpgsql;

CREATE FUNCTION
```

И сам триггер:

```
=> CREATE TRIGGER flights_v_upd_trigger
INSTEAD OF UPDATE ON flights_v
FOR EACH ROW EXECUTE FUNCTION flights_v_update();

CREATE TRIGGER
```

Проверим:

```
=> UPDATE flights_v
SET airport_to = 'Томск. Богашево'
WHERE id = 1;
```

```
UPDATE 1
```

```
=> SELECT * FROM flights_v;
```

```
 id |    airport_from     |   airport_to
----+---------------------+-----------------
  1 | Москва. Шереметьево | Томск. Богашево
(1 row)
```

Попытка изменить аэропорт на отсутствующий в таблице:

```
=> UPDATE flights_v
SET airport_to = 'Южно-Сахалинск. Хомутово'
WHERE id = 1;
```

```
ERROR:  Аэропорт Южно-Сахалинск. Хомутово отсутствует
CONTEXT:  PL/pgSQL function flights_v_update() line 11 at RAISE
```

Событийные триггеры

Событийный триггер

похож на обычный «табличный» триггер, но другой объект

Триггерная функция

соглашение: функция не принимает параметры,
возвращает значение псевдотипа event_trigger

для получения контекста служат специальные функции

События

DDL_COMMAND_START	перед выполнением команды
DDL_COMMAND_END	после выполнения команды
TABLE_REWRITE	перед перезаписью таблицы
SQL_DROP	после удаления объектов

Событийные триггеры — по сути те же триггеры, но срабатывают они не на DML-, а на DDL-операции (CREATE, ALTER, DROP, COMMENT, GRANT, REVOKE).

Такие триггеры являются не инструментом разработки приложений, а скорее служат для решения задач администрирования. Поэтому здесь мы упоминаем их только для полноты картины и рассмотрим только простой пример.

https://postgrespro.ru/docs/postgresql/12/event-trigger-definition

https://postgrespro.ru/docs/postgresql/12/event-trigger-matrix

https://postgrespro.ru/docs/postgresql/12/sql-createeventtrigger

https://postgrespro.ru/docs/postgresql/12/functions-event-triggers

Триггеры событий

Пример триггера для события ddl_command_end, которое соответствует завершению DDL-операции.

Создадим функцию, которая описывает контекст вызова:

```
=> CREATE OR REPLACE FUNCTION describe_ddl() RETURNS event_trigger
AS $$
DECLARE
    r record;
BEGIN
    -- Для события ddl_command_end контекст вызова в специальной функции
    FOR r IN SELECT * FROM pg_event_trigger_ddl_commands()
    LOOP
        RAISE NOTICE '%. тип: %, OID: %, имя: % ',
            r.command_tag, r.object_type, r.objid, r.object_identity;
    END LOOP;
    -- Функции триггера событий не нужно возвращать значение
END;
$$ LANGUAGE plpgsql;
```

```
CREATE FUNCTION
```

Сам триггер:

```
=> CREATE EVENT TRIGGER after_ddl
ON ddl_command_end EXECUTE FUNCTION describe_ddl();
```

```
CREATE EVENT TRIGGER
```

Создаем новую таблицу:

```
=> CREATE TABLE t1(id integer PRIMARY KEY GENERATED ALWAYS AS IDENTITY);
```

```
NOTICE:  CREATE SEQUENCE. тип: sequence, OID: 16895, имя: public.t1_id_seq
NOTICE:  CREATE TABLE. тип: table, OID: 16897, имя: public.t1
NOTICE:  CREATE INDEX. тип: index, OID: 16900, имя: public.t1_pkey
NOTICE:  ALTER SEQUENCE. тип: sequence, OID: 16895, имя: public.t1_id_seq
CREATE TABLE
```

Создание таблицы может может привести к выполнению нескольких команд DDL, поэтому функция pg_event_trigger_ddl_commands возвращает множество строк.

Итоги

Триггер — способ отреагировать на возникновение события

С помощью триггера можно отменить операцию, изменить ее результат или выполнить дополнительные действия

Триггер выполняется как часть транзакции; ошибка в триггере приводит к откату транзакции

Использование триггеров AFTER ROW и переходных таблиц удорожает обработку

Все хорошо в меру: сложную логику трудно отлаживать из-за неявного выполнения триггеров

Практика

1. Создайте триггер, обрабатывающий обновление поля onhand_qty представления catalog_v.

   Проверьте, что в «Каталоге» появилась возможность заказывать книги.

2. Обеспечьте выполнение требования согласованности: количество книг на складе не может быть отрицательным (нельзя купить книгу, которой нет в наличии).

   Внимательно проверьте правильность реализации, учитывая, что с приложением могут одновременно работать несколько пользователей.

2. Может показаться, что достаточно создать AFTER-триггер на таблице operations, подсчитывающий сумму qty_change. Однако на уровне изоляции Read Committed, с которым работает приложение «Книжный магазин», нам придется блокировать таблицу operations в эксклюзивном режиме — иначе возможны сценарии, при которых такая проверка не сработает.

Лучше поступить следующим образом: добавить в таблицу books поле onhand_qty и создать триггер, изменяющий это поле при изменении таблицы operations (то есть, фактически, выполнить денормализацию данных). На поле onhand_qty теперь можно наложить ограничение CHECK, реализующее требование согласованности. А функция onhand_qty(), которую мы создавали ранее, больше не нужна.

Особое внимание надо уделить начальной установке значения, учитывая, что одновременно с выполнением наших операций в системе могут работать пользователи.

1. Триггер для обновления каталога

```
=> CREATE OR REPLACE FUNCTION update_catalog() RETURNS trigger
AS $$
BEGIN
    INSERT INTO operations(book_id, qty_change) VALUES
        (OLD.book_id, NEW.onhand_qty - coalesce(OLD.onhand_qty,0));
    RETURN NEW;
END;
$$ VOLATILE LANGUAGE plpgsql;
```

```
CREATE FUNCTION
```

```
=> CREATE TRIGGER update_catalog_trigger
INSTEAD OF UPDATE ON catalog_v
FOR EACH ROW
EXECUTE FUNCTION update_catalog();
```

```
CREATE TRIGGER
```

2. Проверка количества книг

Добавляем к таблице книг поле наличного количества. (До версии 11 важно было учитывать, что указание предложения DEFAULT вызывало перезапись всех строк таблицы, удерживая блокировку.)

```
=> ALTER TABLE books ADD COLUMN onhand_qty integer;
```

```
ALTER TABLE
```

Триггерная функция для AFTER-триггера на вставку для обновления количества (предполагаем, что поле onhand_qty не может быть пустым):

```
=> CREATE OR REPLACE FUNCTION update_onhand_qty() RETURNS trigger
AS $$
BEGIN
    UPDATE books
    SET onhand_qty = onhand_qty + NEW.qty_change
    WHERE book_id = NEW.book_id;
    RETURN NULL;
END;
$$ VOLATILE LANGUAGE plpgsql;
```

```
CREATE FUNCTION
```

Дальше все происходит внутри транзакции.

```
=> BEGIN;
```

```
BEGIN
```

Блокируем операции на время транзакции:

```
=> LOCK TABLE operations;
```

```
LOCK TABLE
```

Начальное заполнение:

```
=> UPDATE books b
SET onhand_qty = (
    SELECT coalesce(sum(qty_change),0)
    FROM operations o
    WHERE o.book_id = b.book_id
);
```

```
UPDATE 6
```

Теперь, когда поле заполнено, задаем ограничения:

```
=> ALTER TABLE books ALTER COLUMN onhand_qty SET DEFAULT 0;
```

```
ALTER TABLE
```

```
=> ALTER TABLE books ALTER COLUMN onhand_qty SET NOT NULL;
```

```
ALTER TABLE
```

```
=> ALTER TABLE books ADD CHECK(onhand_qty >= 0);
```

```
ALTER TABLE
```

Создаем триггер:

```
=> CREATE TRIGGER update_onhand_qty_trigger
AFTER INSERT ON operations
FOR EACH ROW
EXECUTE FUNCTION update_onhand_qty();

CREATE TRIGGER
```

Готово.

```
=> COMMIT;

COMMIT
```

Теперь books.onhand_qty обновляется, но представление catalog_v по-прежнему вызывает функцию для подсчета количества. Хоть в исходном запросе обращение к функции синтаксически не отличается от обращения к полю, запрос был запомнен в другом виде:

```
=> \d+ catalog_v

                          View "bookstore.catalog_v"
    Column    |  Type   | Collation | Nullable | Default | Storage  | Description
--------------+---------+-----------+----------+---------+----------+-------------
 book_id      | integer |           |          |         | plain    |
 title        | text    |           |          |         | extended |
 onhand_qty   | integer |           |          |         | plain    |
 display_name | text    |           |          |         | extended |
 authors      | text    |           |          |         | extended |
View definition:
 SELECT b.book_id,
    b.title,
    onhand_qty(b.*) AS onhand_qty,
    book_name(b.book_id, b.title) AS display_name,
    authors(b.*) AS authors
   FROM books b
  ORDER BY (book_name(b.book_id, b.title));
Triggers:
    update_catalog_trigger INSTEAD OF UPDATE ON catalog_v FOR EACH ROW EXECUTE FUNCTION update_catalog()
```

Пересоздадим представление:

```
=> CREATE OR REPLACE VIEW catalog_v AS
SELECT b.book_id,
       b.title,
       b.onhand_qty,
       book_name(b.book_id, b.title) AS display_name,
       b.authors
FROM   books b
ORDER BY display_name;

CREATE VIEW
```

Теперь функцию можно удалить.

```
=> DROP FUNCTION onhand_qty(books);

DROP FUNCTION
```

Небольшая проверка:

```
=> SELECT * FROM catalog_v WHERE book_id = 1 \gx

-[ RECORD 1 ]+-----------------------------------
book_id      | 1
title        | Сказка о царе Салтане
onhand_qty   | 19
display_name | Сказка о царе Салтане. Пушкин А. С.
authors      | Пушкин Александр Сергеевич


=> INSERT INTO operations(book_id, qty_change) VALUES (1,+10);

INSERT 0 1

=> SELECT * FROM catalog_v WHERE book_id = 1 \gx

-[ RECORD 1 ]+-----------------------------------
book_id      | 1
title        | Сказка о царе Салтане
onhand_qty   | 29
display_name | Сказка о царе Салтане. Пушкин А. С.
authors      | Пушкин Александр Сергеевич
```

Некорректные операции обрываются:

```
=> INSERT INTO operations(book_id, qty_change) VALUES (1,-100);

ERROR:  new row for relation "books" violates check constraint "books_onhand_qty_check"
DETAIL:  Failing row contains (1, Сказка о царе Салтане, -71).
CONTEXT:  SQL statement "UPDATE books
    SET onhand_qty = onhand_qty + NEW.qty_change
    WHERE book_id = NEW.book_id"
PL/pgSQL function update_onhand_qty() line 3 at SQL statement
```

Практика

1. Напишите триггер, увеличивающий счетчик (поле version) на единицу при каждом изменении строки. При вставке новой строки счетчик должен устанавливаться в единицу. Проверьте правильность работы.
2. Даны таблицы заказов (orders) и строк заказов (lines). Требуется выполнить денормализацию: автоматически обновлять сумму заказа в таблице orders при изменении строк в заказе.

   Создайте необходимые триггеры с использованием переходных таблиц для минимизации операций обновления.

2. Для создания таблиц используйте команды:

```
CREATE TABLE orders (
    id int PRIMARY KEY,
    total_amount numeric(20,2) NOT NULL DEFAULT 0
);

CREATE TABLE lines (
   id int PRIMARY KEY GENERATED ALWAYS AS IDENTITY,
   order_id int NOT NULL REFERENCES orders(id),
   amount numeric(20,2) NOT NULL
);
```

Столбец orders.total_amount должен автоматически вычисляться как сумма значений столбца lines.amount всех строк, относящихся к соответствующему заказу.

1. Счетчик номера версии

```
=> CREATE DATABASE plpgsql_triggers;
```

```
CREATE DATABASE
```

```
=> \c plpgsql_triggers
```

```
You are now connected to database "plpgsql_triggers" as user "student".
```

Таблица:

```
=> CREATE TABLE t(
    id integer PRIMARY KEY GENERATED ALWAYS AS IDENTITY,
    s text,
    version integer
);
```

```
CREATE TABLE
```

Триггерная функция:

```
=> CREATE OR REPLACE FUNCTION inc_version() RETURNS trigger
AS $$
BEGIN
    IF TG_OP = 'INSERT' THEN
        NEW.version := 1;
    ELSE
        NEW.version := OLD.version + 1;
    END IF;
    RETURN NEW;
END;
$$ LANGUAGE plpgsql;
```

```
CREATE FUNCTION
```

Триггер:

```
=> CREATE TRIGGER t_inc_version
BEFORE INSERT OR UPDATE ON t
FOR EACH ROW EXECUTE FUNCTION inc_version();
```

```
CREATE TRIGGER
```

Проверяем:

```
=> INSERT INTO t(s) VALUES ('Раз');
```

```
INSERT 0 1
```

```
=> SELECT * FROM t;
```

```
 id |  s  | version
----+-----+---------
  1 | Раз |       1
(1 row)
```

Явное указание version игнорируется:

```
=> INSERT INTO t(s,version) VALUES ('Два',42);
```

```
INSERT 0 1
```

```
=> SELECT * FROM t;
```

```
 id |  s  | version
----+-----+---------
  1 | Раз |       1
  2 | Два |       1
(2 rows)
```

Изменение:

```
=> UPDATE t SET s = lower(s) WHERE id = 1;
```

```
UPDATE 1
```

```
=> SELECT * FROM t;
```

```
 id |  s  | version
----+-----+---------
  2 | Два |       1
  1 | раз |       2
(2 rows)
```

Явное указание также игнорируется:

```
=> UPDATE t SET s = lower(s), version = 42 WHERE id = 2;
```

```
UPDATE 1
```

```
=> SELECT * FROM t;
```

```
 id |  s  | version
----+-----+---------
  1 | раз |       2
  2 | два |       2
(2 rows)
```

2. Автоматическое вычисление общей суммы заказов

Создаем таблицы упрощенной структуры, достаточной для демонстрации:

```
=> CREATE TABLE orders (
    id integer PRIMARY KEY,
    total_amount numeric(20,2) NOT NULL DEFAULT 0
);
```

```
CREATE TABLE
```

```
=> CREATE TABLE lines (
   id integer PRIMARY KEY GENERATED ALWAYS AS IDENTITY,
   order_id integer NOT NULL REFERENCES orders(id),
   amount numeric(20,2) NOT NULL
);
```

```
CREATE TABLE
```

Создаем триггерную функцию и триггер для обработки вставки:

```
=> CREATE FUNCTION total_amount_ins() RETURNS trigger
AS $$
BEGIN
    WITH l(order_id, total_amount) AS (
        SELECT order_id, sum(amount)
        FROM new_table
        GROUP BY order_id
    )
    UPDATE orders o
    SET total_amount = o.total_amount + l.total_amount
    FROM l
    WHERE o.id = l.order_id;
    RETURN NULL;
END;
$$ LANGUAGE plpgsql;
```

```
CREATE FUNCTION
```

Предложение FROM в команде UPDATE позволяет соединить orders с подзапросом по переходной таблице и использовать столбцы подзапроса для вычисления значения.

```
=> CREATE TRIGGER lines_total_amount_ins
AFTER INSERT ON lines
REFERENCING
    NEW TABLE AS new_table
FOR EACH STATEMENT
EXECUTE FUNCTION total_amount_ins();
```

```
CREATE TRIGGER
```

Функция и триггер для обработки обновления:

```
=> CREATE FUNCTION total_amount_upd() RETURNS trigger
AS $$
BEGIN
    WITH l_tmp(order_id, amount) AS (
        SELECT order_id, amount FROM new_table
        UNION ALL
        SELECT order_id, -amount FROM old_table
    ), l(order_id, total_amount) AS (

        SELECT order_id, sum(amount)
        FROM l_tmp
        GROUP BY order_id
        HAVING sum(amount) <> 0
    )
    UPDATE orders o
    SET total_amount = o.total_amount + l.total_amount
    FROM l
    WHERE o.id = l.order_id;
    RETURN NULL;
END;
$$ LANGUAGE plpgsql;
```

```
CREATE FUNCTION
```

Условие HAVING позволяет пропускать изменения, не влияющие на общую сумму заказа.

```
=> CREATE TRIGGER lines_total_amount_upd
AFTER UPDATE ON lines
REFERENCING
    OLD TABLE AS old_table
    NEW TABLE AS new_table
FOR EACH STATEMENT
EXECUTE FUNCTION total_amount_upd();
```

```
CREATE TRIGGER
```

Функция и триггер для обработки удаления:

```
=> CREATE FUNCTION total_amount_del() RETURNS trigger
AS $$
BEGIN
    WITH l(order_id, total_amount) AS (
        SELECT order_id, -sum(amount)
        FROM old_table
        GROUP BY order_id
    )
    UPDATE orders o
    SET total_amount = o.total_amount + l.total_amount
    FROM l
    WHERE o.id = l.order_id;
    RETURN NULL;
END;
$$ LANGUAGE plpgsql;
```

```
CREATE FUNCTION
```

```
=> CREATE TRIGGER lines_total_amount_del
AFTER DELETE ON lines
REFERENCING
    OLD TABLE AS old_table
FOR EACH STATEMENT
EXECUTE FUNCTION total_amount_del();
```

```
CREATE TRIGGER
```

Остался неохваченным оператор TRUNCATE. Однако триггер для этого оператора не может использовать переходные таблицы. Но мы знаем, что после выполнения TRUNCATE в lines не останется строк, значит можно обнулить суммы всех заказов.

```
=> CREATE FUNCTION total_amount_truncate() RETURNS trigger
AS $$
BEGIN
    UPDATE orders SET total_amount = 0;
    RETURN NULL;
END;
$$ LANGUAGE plpgsql;
```

```
CREATE FUNCTION
```

```
=> CREATE TRIGGER lines_total_amount_truncate
AFTER TRUNCATE ON lines
FOR EACH STATEMENT
EXECUTE FUNCTION total_amount_truncate();
```

```
CREATE TRIGGER
```

Дополнительно нужно запретить изменять значение total_amount вручную, но это задача решается не триггерами.

Проверяем работу.

Добавили два новых заказа без строк:

```
=> INSERT INTO orders VALUES (1), (2);

INSERT 0 2

=> SELECT * FROM orders ORDER BY id;

 id | total_amount
----+--------------
  1 |         0.00
  2 |         0.00
(2 rows)
```

Добавили строки в заказы:

```
=> INSERT INTO lines (order_id, amount) VALUES
    (1,100), (1,100), (2,500), (2,500);

INSERT 0 4

=> SELECT * FROM lines;

 id | order_id | amount
----+----------+--------
  1 |        1 | 100.00
  2 |        1 | 100.00
  3 |        2 | 500.00
  4 |        2 | 500.00
(4 rows)


=> SELECT * FROM orders ORDER BY id;

 id | total_amount
----+--------------
  1 |       200.00
  2 |      1000.00
(2 rows)
```

Удвоили суммы всех строк всех заказов:

```
=> UPDATE lines SET amount = amount * 2;

UPDATE 4

=> SELECT * FROM orders ORDER BY id;

 id | total_amount
----+--------------
  1 |       400.00
  2 |      2000.00
(2 rows)
```

Удалим одну строку первого заказа:

```
=> DELETE FROM lines WHERE id = 1;

DELETE 1

=> SELECT * FROM orders ORDER BY id;

 id | total_amount
----+--------------
  1 |       200.00
  2 |      2000.00
(2 rows)
```

Опустошим таблицу строк:

```
=> TRUNCATE lines;

TRUNCATE TABLE

=> SELECT * FROM orders ORDER BY id;
```

```
 id | total_amount
----+--------------
  1 |         0.00
  2 |         0.00
(2 rows)
```