**Триггеры.**

**Триггеры** PostgreSQL – это функции обратного вызова базы данных, которые автоматически выполняются / запускаются, когда происходит указанное событие базы данных. Триггер PostgreSQL может быть указан для запуска

* Перед попыткой операции над строкой (перед проверкой ограничений и попыткой INSERT, UPDATE или DELETE)
* После завершения операции (после проверки ограничений и завершения INSERT, UPDATE или DELETE)
* Вместо операции (в случае вставки, обновления или удаления в представлении)

Триггер, помеченный FOR EACH ROW, вызывается один раз для каждой строки, которую изменяет операция. Напротив, триггер, помеченный FOR EACH STATEMENT, выполняется только один раз для любой данной операции, независимо от того, сколько строк он изменяет.

|  |  |
| --- | --- |
| ПРОВЕРЬТЕ РЕЗУЛЬТАТ РАБОТЫ ПРОГРАММЫ  Убедитесь, что в таблицу EMPLOYEES нельзя вносить изменения до 6 часов утра и после 10 часов вечера в течение дня.  create or replace trigger trg\_employees\_time\_check  before update or insert or delete  on employees  for each row  begin  if to\_char(sysdate,'hh24') < 6 or to\_char(sysdate,'hh24') > 10 then  raise\_application\_error(-20111,'Sorry! No change can be made before 6 AM and after 10 PM');  end if;  end;  ПРОВЕРЬТЕ РЕЗУЛЬТАТ РАБОТЫ ТРИГГЕРА  Создайте триггер, чтобы заработная плата работника не уменьшалась.  create or replace trigger trg\_employees\_salary\_check  before update  on employees  for each row  begin  if :old.salary > :new.salary then  raise\_application\_error(-20111,'Sorry! Salary can not be decreased!');  end if;  end;  ПРОВЕРЬТЕ РЕЗУЛЬТАТ РАБОТЫ ТРИГГЕРА  SQL>update employees set salary = 10  where department\_id = 10;  Создайте триггер, чтобы сотрудник и менеджер принадлежали к одному отделу.  Примечание: этот триггер должен прочитать строку, которая изменяется, что вызывает проблему с мутацией.  create or replace trigger trg\_log\_job\_change  after update of job\_id  on employees  for each row  declare  v\_enddate date;  v\_startdate date;  begin  -- find out whether the employee has any row in job\_history table  select max(end\_date) into v\_enddate  from job\_history  where employee\_id = :old.employee\_id;  if v\_enddate is null then  v\_startdate := :old.hire\_date;  else  v\_startdate := v\_enddate + 1;  end if;  insert into job\_history values (:old.employee\_id, v\_startdate, sysdate, :old.job\_id, :old.department\_id);  end;   ПРОВЕРЬТЕ РЕЗУЛЬТАТ РАБОТЫ ТРИГГЕРА | PostgreSQL  C  CREATE OR REPLACE FUNCTION time\_check\_trigger\_fnc()  RETURNS trigger AS  $$  DECLARE  cur\_hour numeric;  BEGIN  SELECT EXTRACT(HOUR FROM CURRENT\_TIMESTAMP) INTO cur\_hour;  If cur\_hour < 6 OR cur\_hour > 22  THEN  RAISE EXCEPTION 'Sorry! No change can be made before 6 AM and after 10 PM';  END IF;  RETURN NEW;  End;  $$  LANGUAGE 'plpgsql';  CREATE TRIGGER time\_trigger  BEFORE UPDATE  ON "employees"  FOR EACH ROW  EXECUTE PROCEDURE time\_check\_trigger\_fnc();  CREATE OR REPLACE FUNCTION salary\_no\_decrease\_check\_trigger\_fnc()  RETURNS trigger AS  $$  BEGIN  If OLD.salary > NEW.salary  THEN  RAISE EXCEPTION 'Sorry! Salary can not be decreased ';  END IF;  RETURN NEW;  End;  $$  LANGUAGE 'plpgsql';  CREATE TRIGGER salary\_trigger  BEFORE UPDATE  ON "employees"  FOR EACH ROW  EXECUTE PROCEDURE salary\_no\_decrease\_check\_trigger\_fnc ();  CREATE OR REPLACE FUNCTION history\_trigger\_fnc()  RETURNS trigger AS  $$  DECLARE  v\_enddate timestamp;  v\_startdate timestamp;  BEGIN  select max(end\_date) into v\_enddate  from job\_history  where employee\_id = OLD.employee\_id;  if v\_enddate is null then  v\_startdate := OLD.hire\_date;  else  v\_startdate := v\_enddate + INTERVAL '1 day';  end if;  insert into job\_history values  (OLD.employee\_id, v\_startdate, CURRENT\_DATE::timestamp, OLD.job\_id, OLD.department\_id);  RETURN NEW;  End;  $$  LANGUAGE 'plpgsql';  CREATE OR REPLACE TRIGGER job\_history\_trigger  AFTER UPDATE OF job\_id  ON "employees"  FOR EACH ROW  EXECUTE PROCEDURE history\_trigger\_fnc(); |
|  |  |

PostgreSQL-триггеры: создание, удаление, примеры

Триггеры используют для того, чтобы сказать движку PostgreSQL выполнить часть кода при наступлении определённого события. Получается своего рода катализатор изменений, спусковой крючок, который запускает цепь событий.

Триггер должен быть связан с указанной таблицей, представлением (псевдотаблицей) или внешней таблицей. Он запускает свою часть кода только при выполнении операций с этой сущностью — INSERT, UPDATE, DELETE или TRUNCATE. В зависимости от требований мы можем запускать триггер до, после или вместо события/операции.

**Типы триггеров**

Триггеры делятся на два типа в зависимости от того, на каком уровне они действуют.

Если триггер помечен опцией FOR EACH ROW, тогда функция вызывается для каждой строки, которая изменяется в результате события. Например, если сделать UPDATE для 100 строк, триггерная функция UPDATE будет вызываться 100 раз, по одному разу для каждой обновлённой строки.

Опция FOR EACH STATEMENT вызовет функцию только один раз для каждого оператора, независимо от количества изменяемых строк.

**Использование триггеров**

Это довольно мощный инструмент, у которого много сценариев использования. Вот лишь несколько примеров:

1. Вы можете использовать триггерные конструкции для отслеживания транзакций таблицы, регистрируя сведения о событии.
2. Вы можете создать триггер, с помощью которого будете проверять ограничения перед применением транзакции.
3. С помощью таких спусковых крючков вы можете автоматически заполнять поля, используя записи новых транзакций.

Триггеры помогают оптимизировать количество запросов. Например, у вас на сервере [Timeweb Cloud](https://timeweb.cloud/) есть таблица, в которую записываются временные метки. Задача — агрегировать данные за указанные интервалы (пусть их будет четыре в сутки, каждый продолжительностью 6 часов).

Если каждый раз сканировать таблицу, выполнять группировку, сортировку и все расчёты (допустим, вычисление среднего значения), то на больших данных быстро станет заметной неэффективность работы — не помогут даже мощные [облачные серверы](https://timeweb.cloud/vds-vps).

Чтобы не обрабатывать все данные каждый раз заново, можно использовать [Materialized Views](https://www.postgresql.org/docs/current/rules-materializedviews.html) — это представления, которые сохраняют результаты в табличной форме. Они позволяют закэшировать данные. Проблема в том, что при каждом обновлении представление пересчитывается целиком. На больших данных это снова может стать проблемой.

Здесь на помощь и приходит триггер. Он позволяет создать по сути тот же Materialized View, только умный. Он не пересчитывает все данные, а обновляет только ту строку, в которую внесли изменения.

**Создание триггера**

С практической пользой от использования разобрались. Теперь посмотрим, как создать триггер в PostgreSQL.

Синтаксис запроса следующий:

CREATE [ OR REPLACE ] [ CONSTRAINT ] TRIGGER name { BEFORE | AFTER | INSTEAD OF } { event [ OR ... ] }

    ON table\_name

    [ FROM referenced\_table\_name ]

   [ NOT DEFERRABLE | [ DEFERRABLE ] [ INITIALLY IMMEDIATE | INITIALLY DEFERRED ] ]

    [ REFERENCING { { OLD | NEW } TABLE [ AS ] transition\_relation\_name } [ ... ] ]

    [ FOR [ EACH ] { ROW | STATEMENT } ]

    [ WHEN ( condition ) ]

    EXECUTE { FUNCTION | PROCEDURE } function\_name ( arguments )

где событие (event) может быть одним из следующих:

    INSERT

    UPDATE [ OF column\_name [, ... ] ]

    DELETE

    TRUNCATE

Здесь требуется несколько пояснений.

1. Вы можете создать (CREATE) или заменить (REPLACE) уже существующий триггер.
2. Вы сразу связываете функцию с конкретной таблицей, представлением или внешней таблицей. Код будет исполняться только при наступлении события с этой связанной сущностью.
3. Триггеры с опцией INSTEAD OF должны быть помечены опцией FOR EACH ROW и могут быть определены только в представлениях. Триггеры, которые выполняются до (BEFORE) или после события (AFTER) в представлении должны быть помечены как FOR EACH STATEMENT. В [документации](https://www.postgresql.org/docs/14/sql-createtrigger.html) есть таблица, которая поможет сориентироваться.

**Простые примеры**

Чтобы разобраться с синтаксисом, посмотрим на примеры триггеров PostgreSQL.

Например, здесь вы говорите движку, что нужно выполнять функцию check\_account\_update() каждый раз до обновления таблицы accounts:

CREATE TRIGGER check\_update

    BEFORE UPDATE ON accounts

    FOR EACH ROW

    EXECUTE FUNCTION check\_account\_update();

В этом примере вы устанавливаете дополнительное условие. Функция должна выполняться только в том случае, если обновляется столбец balance в таблице accounts.

CREATE OR REPLACE TRIGGER check\_update

    BEFORE UPDATE OF balance ON accounts

    FOR EACH ROW

    EXECUTE FUNCTION check\_account\_update();

А это триггер для добавления записей в журнал. Функция срабатывает только после того, как в таблицу accounts внесли изменения:

CREATE TRIGGER log\_update

    AFTER UPDATE ON accounts

    FOR EACH ROW

    WHEN (OLD.\* IS DISTINCT FROM NEW.\*)

    EXECUTE FUNCTION log\_account\_update();

Ещё один пример — с INSTEAD OF. Функция view\_insert\_row() выполняется для каждой строки, чтобы вставить строки в таблицы, лежащие в основе представления:

CREATE TRIGGER view\_insert

    INSTEAD OF INSERT ON my\_view

    FOR EACH ROW

    EXECUTE FUNCTION view\_insert\_row();

Триггер на удаление в PostgreSQL можно добавить к транзакциям, удаляющим записи:

CREATE TRIGGER example\_delete\_trigger

  AFTER DELETE   ON my\_view

  FOR EACH ROW

 EXECUTE PROCEDURE aft\_delete();

**Практика — добавление информации в две таблицы**

Давайте рассмотрим пример создания триггера PostgreSQL, который будет добавлять в таблицу информацию о новом сотруднике, если эти данные появились в другой таблице.

Сначала нужно создать обе таблицы:

CREATE TABLE "Employee"

(

"EmployeeId" INT NOT NULL,

"LastName" VARCHAR(20) NOT NULL,

"FirstName" VARCHAR(20) NOT NULL,

"Title" VARCHAR(30),

"ReportsTo" INT,

"BirthDate" TIMESTAMP,

"HireDate" TIMESTAMP,

"Address" VARCHAR(70),

"City" VARCHAR(40),

"State" VARCHAR(40),

"Country" VARCHAR(40),

"PostalCode" VARCHAR(10),

"Phone" VARCHAR(24),

"Fax" VARCHAR(24),

"Email" VARCHAR(60),

CONSTRAINT "PK\_Employee" PRIMARY KEY  ("EmployeeId")

);

CREATE TABLE "Employee\_Audit"

(

"EmployeeId" INT NOT NULL,

"LastName" VARCHAR(20) NOT NULL,

"FirstName" VARCHAR(20) NOT NULL,

"UserName" VARCHAR(20) NOT NULL,

"EmpAdditionTime" VARCHAR(20) NOT NULL,

);

Таблицы готовы, теперь нужно добавить триггерную функцию, чтобы настроить между ними обмен данными по наступлению события. В нашем случае событие — это добавление информации о новом сотруднике в таблицу «Employee».

CREATE OR REPLACE FUNCTION employee\_insert\_trigger\_fnc()

  RETURNS trigger AS

$$

BEGIN

 INSERT INTO "Employee\_Audit" ( "EmployeeId", "LastName", "FirstName","UserName" ,"EmpAdditionTime")

VALUES(NEW."EmployeeId",NEW."LastName",NEW."FirstName",current\_user,current\_date);

RETURN NEW;

END;

$$

LANGUAGE 'plpgsql';

CREATE TRIGGER employee\_insert\_trigger

  AFTER INSERT

  ON "Employee"

  FOR EACH ROW

EXECUTE PROCEDURE employee\_insert\_trigger\_fnc();

Как только мы выполним описанный выше INSERT в «Employee», триггер добавит одну новую запись в «Employee\_Audit» со следующими данными:

INSERT INTO "Employee"

VALUES(12,' Smith','Jeff','Editor',1,'1992-05-28 00:00:00','2022-01-15 00:00:00','Paseo de Gracia','Barcelona','Catalonia','Spain','128 665','+15 52-469-2573','+15 52-469-2573','mail@mail.com')

Теперь проверим, что всё работает так, как мы предполагали. Сначала выведем сведения о сотруднике из таблицы «Employee», в которую мы только что вставили данные:

SELECT \* FROM "Employee" WHERE "EmployeeId" =12;

EmployeeId | 12

LastName   | Smith

FirstName  | Jeff

Title  | Editor

ReportsTo  | 1

BirthDate  | 1992-05-28 00:00:00

HireDate   | 2022-01-15 00:00:00

Address | Paseo de Gracia

City   | Barcelona

State  | Catalonia

Country | Spain

PostalCode | 128 665

Phone  | +15 52-469-2573

Fax    | +15 52-469-2573

Email  | mail@mail.com

Теперь посмотрим, записались ли нужные данные в таблицу «Employee\_Audit»:

SELECT \* FROM "Employee\_Audit" ;

EmployeeId   | 12

LastName    | Smith

FirstName   | Jeff

UserName    | postgres

EmpAdditionTime | 2022-06-17

Отлично, всё работает!

**Изменение триггера**

Чтобы изменить свойства триггера, используйте CREATE OR REPLACE TRIGGER, указав имя существующей триггерной функции и связанную таблицу. Остальные свойства вы можете менять так, как нужно для выполнения вашей задачи.

Вы также можете переименовать триггер. Для этого используйте запрос ALTER TRIGGER:

ALTER TRIGGER name ON table\_name RENAME TO new\_name

Подробности [смотрите в документации](https://www.postgresql.org/docs/14/sql-altertrigger.html).

**Удаление триггера**

Используйте DROP TRIGGER, чтобы удалить триггер PostgreSQL. Синтаксис очень простой:

DROP TRIGGER [ IF EXISTS ] name ON table\_name [ CASCADE | RESTRICT ]

Например, так вы удалите some\_example\_of\_trigger, связанный с таблицей Example:

DROP TRIGGER some\_example\_of\_trigger ON "Example" ;

Для возможности удаления триггера пользователь должен быть владельцем таблицы.

Можно использовать дополнительные параметры при отключении:

* IF EXISTS — указание на то, что не надо выдавать ошибку, если такого триггера нет.
* CASCADE — автоматически удалять все объекты, которые зависят от триггера, объекты, которые зависят от этих объектов, и так далее.
* RESTRICT — не удалять триггер, если от него зависят другие объекты. Это значение по умолчанию.

Полное описание [смотрите в документации](https://www.postgresql.org/docs/14/sql-droptrigger.html).

**Важные моменты, которые следует помнить**

1. Чтобы создать триггер, пользователь должен иметь привилегию TRIGGER для таблицы и привилегию EXECUTE для функции.
2. Вы можете проверить системный каталог «pg\_trigger» на наличие существующей информации о триггерах в базе данных.
3. Если вы создадите несколько триггеров для одного и того же объекта и для одного и того же события, они будут срабатывать в алфавитном порядке по имени.