Кафедра ЦТ
Институт информационных технологий
РТУ МИРЭА

Дисциплина «Разработка баз данных»

Практическая работа №6. Триггеры и курсоры в PostgreSQL

Постановка задачи: основываясь на индивидуальной схеме данных, составьте необходимые запросы:

Задание №1: создание триггеров

Проанализировать предметную область своей базы данных и выявить не менее **трёх бизнес-правил**, реализация которых в виде ограничений целостности **возможна** только **с помощью триггеров**.

Для **КАЖДОГО** **правила создать триггер** (всего **ТРИ ТРИГГЕРА**):

- **описать алгоритм** его работы, указав таблицу, событие и последовательность действий;
- **написать код** триггерной **функции** на PL/pgSQL и **оператора** **CREATE TRIGGER**;
- **продемонстрировать работу** триггера на **примерах DML-операций**, которые как **успешно выполняются**, так и **корректно** **прерываются** триггером (**два** **запроса** на **каждый** **триггер**).

ВАЖНО: **триггеры** должны быть **разными**.

(продолжение на следующем слайде)

Практическая работа №6. Триггеры и курсоры в PostgreSQL

Постановка задачи: основываясь на индивидуальной схеме данных, составьте необходимые запросы:

Задание №2: создание курсоров

Разработать два скрипта на PL/pgSQL, демонстрирующих оба способа обработки данных.

- **Скрипт 1**: с использованием **Явного курсора**.
  Должен включать **DECLARE**, **OPEN**, **FETCH** в цикле **LOOP** и **CLOSE**.
- **Скрипт 2**: с использованием **Неявного курсора**.
  Должен использовать цикл **FOR**...**IN**.

ВАЖНО: если Ваша база данных **не содержит** достаточно **таблиц** и/или **полей** для выполнения задания или его части – значит, необходимо **доработать базу**.

Данная проблема **не является основанием** для **пропуска** какого-либо задания или его части.

(продолжение на предыдущем слайде)

ТРИГГЕРЫ (TRIGGERS)

Триггеры (Triggers) – зачем нужны триггеры?

Стандартные *(декларативные)* **ограничения** (**PRIMARY KEY**, **FOREIGN KEY**, **CHECK**) **мощные**, но **ограниченные**.

Чего они НЕ могут?

- **Проверять данные в других таблицах**: **CHECK** не может **«видеть» данные** в **других таблицах**.
  (Например, проверить остаток на складе перед продажей).
- **Выполнять сложные вычисления**: логика **CHECK** ограничена **простыми выражениями**.
- **Выполнять побочные действия**: ограничения не могут записывать **логи** или **изменять данные** в **других таблицах**.

Решением стали Триггеры

Они позволяют выполнить **произвольный код** в ответ на DML-операции (**INSERT**, **UPDATE**, **DELETE**).

Триггеры (Triggers) – что такое триггер?

Триггер – это **функция**, которая **автоматически запускается** в ответ на определённое **DML-событие в таблице**, **к которой он привязан**.

Ключевые компоненты:

- **Событие (Event)** – ***условие активации***. Может быть **INSERT** , **UPDATE** , **DELETE** , **TRUNCATE**.
- **Время (Timing)** – ***когда он срабатывает?***
  - ✓ **BEFORE** – **до выполнения операции**. Позволяет проверить или изменить данные.
  - ✓ **AFTER** – **после выполнения операции**. Используется для действий, зависящих от успешного завершения.
  - ✓ **INSTEAD OF** – **вместо операции**. Используется только для представлений (**VIEW**).
- **Уровень (Level)** – ***как часто он срабатывает?***
  - ✓ **FOR EACH ROW** – **один раз для каждой строки**, затронутой операцией.
  - ✓ **FOR EACH STATEMENT** – **один раз за всю DML-операцию** *(независимо от того, сколько строк изменено)*.

Триггеры (Triggers) – «магические» переменные: NEW и OLD

Это **переменные типа RECORD**, содержащие **данные строки до** и **после операции**.

Доступны **только** в триггерах уровня **FOR EACH ROW**.

Операция	NEW *(данные ПОСЛЕ операции)*	OLD *(данные ДО операции)*
INSERT	Содержит **ВСТАВЛЯЕМУЮ** строку	**NULL** *(строки ещё не было)*
UPDATE	Содержит **НОВУЮ** версию строки	Содержит **СТАРУЮ** версию строки
DELETE	**NULL** *(строки больше не будет)*	Содержит **УДАЛЯЕМУЮ** строку

ВАЖНО: в **BEFORE** триггерах можно **изменять значения** в **NEW**.

```
NEW.название_столбца := новое_значение;
```

Триггеры (Triggers) – логика триггера: BEFORE vs AFTER

Используйте BEFORE *(до операции)* для задач:

- ✓ **Валидация** – проверка данных до их попадания в таблицу.
  Пример: *Хватит ли товара на складе?*
  ВАЖНО: если данные не верны, можно **отменить операцию** через **RAISE EXCEPTION**.
- ✓ **Модификация** – изменение данных «на лету».
  Пример*: привести* ***email*** *к нижнему регистру:* `NEW.email = LOWER(NEW.email);`

Используйте AFTER *(после операции)* для задач:

- ✓ **Аудит и Логирование** – запись действий, которые уже успешно произошли.
  Пример: записать в ***customer_audit***, что телефон был изменён.
- ✓ **Обновление связанных таблиц** (согласованность)
  Пример: после **INSERT** в ***sale_items***, пересчитать ***total_amount*** в ***sales***.
- ✓ **Нюанс – отмена через RAISE EXCEPTION**
  RAISE EXCEPTION в **AFTER** триггере **тоже отменит** создание записи, вызвав откат (**ROLLBACK**) всей транзакции.
  Это сработает, так как **любая SQL-команда** *(даже один* ***INSERT****)* **выполняется в транзакции** *(явной или неявной)*.

Триггеры (Triggers) – логика триггера: ROW vs STATEMENT

FOR EACH ROW *(уровень строки)* – выполняется один раз **для каждой строки**, затронутой DML-операцией.

Пример: `UPDATE ... WHERE price > 10;` *(изменил **50 строк**)* => **триггер** сработает **50 раз**.

✓ **Имеет доступ** к специальным переменным **NEW** и **OLD**.

✓ Используется в **99% случаев**.

FOR EACH STATEMENT *(уровень оператора)* – выполняется **один раз за всю DML-операцию**, **независимо** от количества измененных строк.

Пример: `UPDATE ... WHERE price > 10;` *(изменил **50 строк**)* => **триггер** сработает **1 раз**.

✓ Это – **значение по умолчанию**.

✓ **Не имеет доступа** к **NEW** и **OLD**.

✓ Используется для **общего аудита** *(«таблица X была изменена пользователем Y»)*, или **сложных проверок**, не зависящих от конкретных строк.

Триггеры (Triggers) – управление операцией

В **BEFORE-триггерах** уровня **ROW** возвращаемое значение решает судьбу **каждой отдельной строки**.

В **AFTER-триггерах** оно **игнорируется**, но указывать `RETURN NULL|OLD|NEW;` **необходимо для совместимости**.

Разрешение операции *(стандартное поведение)*:

- `RETURN NEW;` – используется для **INSERT** и/или **UPDATE**, чтобы **разрешить запись новой версии** строки.
- `RETURN OLD;` – используется для **DELETE**, чтобы **разрешить удаление старой версии** строки.
- `RETURN NULL;` – **молчаливая отмена** *(**пропуск строки**)* заставляет PostgreSQL **пропустить** **DML-операцию** (**INSERT**, **UPDATE** или **DELETE**) только **для этой строки**, не вызывая ошибки и продолжая транзакцию.

`RAISE EXCEPTION '...';` – **полная отмена** *(для всех)* немедленно прерывает и откатывает **всю транзакцию**.

Нюанс UPDATE: чтобы **запретить изменение** конкретного поля *(например, **created_at**)*, не возвращайте `RETURN OLD;`

Правильный способ: принудительно исправить **NEW** – `NEW.created_at := OLD.created_at;` и вернуть `RETURN NEW;`.

Триггеры (Triggers) – специальная переменная: TG_OP

TG_OP – это **специальная переменная**, доступная **в триггерных функциях**.

Она содержит **текстовую строку**, указывающую, **какая DML-операция** **вызвала триггер**.

Возможные значения:

- `'INSERT'`
- `'UPDATE'`
- `'DELETE'`

Это позволяет использовать **одну и ту же функцию** для **нескольких событий** (***например***, ***AFTER INSERT OR UPDATE OR DELETE***).

```
IF (TG_OP = 'DELETE') THEN ...
ELSIF (TG_OP = 'INSERT') THEN ...
END IF;
```

Триггеры (Triggers) – разделение ответственности

В **PostgreSQL** принято **разделение ответственности**: триггер – это событие, а функция – это логика.

- **Триггерная функция *(логика)*** – содержит сам **код** на **PL/pgSQL**.

  Описывает, **ЧТО** именно нужно сделать *(проверить, залогировать, изменить)*.

  ```
  CREATE FUNCTION ... RETURNS TRIGGER
  ```

- **Триггер *(событие)*** – **привязывает** функцию к таблице.

  Описывает, **КОГДА** нужно запустить логику *(на какое событие, до или после, для каждой строки)*.

  ```
  CREATE TRIGGER ... EXECUTE FUNCTION ...
  ```

Триггеры (Triggers) – создание триггерной функции

Сначала описывается **логика** в **специальной функции**, возвращающей тип **TRIGGER**.

```
CREATE [OR REPLACE] FUNCTION <func_name>()
RETURNS TRIGGER AS $$

DECLARE
    [<переменные>]
BEGIN
    <логика триггера: IF, NEW, OLD, TG_OP>
    RETURN { NEW|OLD|NULL };
END;
$$ LANGUAGE plpgsql;
```

CREATE FUNCTION func_name – создаёт функцию, которая будет содержать логику.

RETURNS TRIGGER – **(обязательно)** указывает, что эта функция предназначена только для **вызова триггером**.

DECLARE – секция **объявления переменных**.

BEGIN ... END – тело функции, где описывается что делать *(проверять, логировать, изменять **NEW** и т.д.)*.

NEW, **OLD** – специальные переменные, хранящие изменяемую и новую строку в рамках операции.

RETURN [**NEW** | **OLD** | **NULL**] – **(важно)** решает, что делать с DML-операцией
(подробности на слайде «управление операцией»).

Триггеры (Triggers) – создание триггера

Затем **событие** *(например,* ***UPDATE*** *таблицы)* **привязывается к созданной функции**.

```
CREATE [OR REPLACE] TRIGGER <trigger_name>
{ BEFORE | AFTER }
{ INSERT [OR] UPDATE [OR] DELETE }
ON <table_name>
[ FOR EACH { ROW | STATEMENT } ]
EXECUTE FUNCTION <func_name>();
```

CREATE TRIGGER trigger_name – создаёт **сам триггер** с уникальным именем.

{BEFORE | AFTER} – **(обязательно)** указывает, когда сработать: **до** или **после** выполнения основного запроса.

{INSERT | UPDATE | ...} – **(обязательно) DML-операция**, которая **активирует триггер**. Можно несколько сразу.

ON table_name – **таблица**, за которой **«следит» триггер**.

FOR EACH ROW – выполнять для каждой изменённой строки *(даёт доступ к* ***NEW/OLD****)*.

FOR EACH STATEMENT – выполнять **один раз** за всю операцию. **Значение по умолчанию!**

EXECUTE FUNCTION – привязывает событие к ранее созданной логике.

КУРСОРЫ (CURSORS)

Курсоры (Cursors) – зачем нужны курсоры?

SQL – **язык декларативный** *(мы говорим, «ЧТО» хотим, а не «КАК» это получить)* и ориентирован на работу со **множествами** данных.

Мы говорим: *«дай мне всех пользователей, у которых страна = 'Россия'»* *и СУБД возвращает нам их* ***всех разом***.

Однако иногда задачи требуют ***процедурного подхода*** – выполнения **сложных действий** **для каждой строки** результата **по очереди**.

Курсор – это и есть решение.

Представьте, что ваш **SELECT-запрос** вернул **1000 строк**.

Курсор – это как **закладка** или **указатель** **на текущую строку** в этом наборе.

Вы можете **«открыть»** этот набор и **в цикле** говорить: *«дай мне строку,* ***на которой закладка****»* (**FETCH**), обработать её, а затем сказать *«передвинь закладку* ***на следующую строку****»*.

Курсоры (Cursors) –
метод 1: Явный курсор (Ручной режим)

Этот способ **даёт полный контроль** над процессом.

```
DO $$
DECLARE
    <cursor_name> CURSOR FOR
        SELECT id, name FROM medicines;
    <row_var> RECORD;
BEGIN
    OPEN <cursor_name>;
    LOOP
        FETCH <cursor_name> INTO <row_var>;
        EXIT WHEN NOT FOUND;
        <Действия с текущей строкой>
    END LOOP;
CLOSE <cursor_name>;
END;
$$ LANGUAGE plpgsql;
```

DECLARE – секция **объявления переменных**.

CURSOR FOR SELECT - **связь переменной-курсора с запросом**.

RECORD – **переменная** для **текущей строки** в цикле.

OPEN – выполняем SQL-запрос, готовим курсор к чтению.

LOOP ... END LOOP – создаёт бесконечный **цикл**.

FETCH ... INTO ... – **извлечение текущей строки** в переменную.

EXIT WHEN – **выходим** из цикла **по условию**.

NOT FOUND – **условие** выхода из цикла – **строки закончились**.

CLOSE – освобождаем ресурсы, связанные с курсором.

Курсоры (Cursors) – метод 2: Неявный курсор (FOR...IN)

Это **предпочтительный**, более **чистый** и **безопасный** способ для большинства задач.

PL/pgSQL автоматически управляет **всем жизненным циклом курсора** (**OPEN**, **FETCH**, **CLOSE**).

Вам **не нужно писать** **DECLARE CURSOR**, **OPEN**, **FETCH** и **CLOSE** – цикл **FOR...IN** делает всё это за вас.

```
DO $$
DECLARE
    <row_var> RECORD;
BEGIN
    FOR <row_var> IN
        SELECT * FROM medicines
    LOOP
        <Действия с текущей строкой>
    END LOOP;
END;
$$;
```

DECLARE – секция **объявления переменных**.

RECORD – **переменная** для **текущей строки** в цикле.

FOR <переменная> IN <запрос> – каждая строка из результатов **<запроса>** будет по очереди помещена в **<переменную>**.

LOOP ... END LOOP – создаёт **цикл** для действий со **<строкой>**.

Обращение к полю в строке – `row_record.name`

Курсоры (Cursors) – цикл курсора: что можно, нельзя, или не следует

Внутри цикла *(в **вашей переменной**)* у вас есть **данные текущей строки** для **процедурной обработки**.

Можно делать практически что угодно:

- **производить сложные вычисления** (**IF/CASE**);
- **изменять другие таблицы** (**INSERT/UPDATE**);
- **вызывать функции** или **готовить данные** для возврата из функции (**RETURN NEXT**).

Вот, что делать нельзя или не следует:

- **Не управляйте транзакцией.** Категорически **нельзя** использовать **COMMIT** или **ROLLBACK** внутри цикла! Это немедленно закроет ваш курсор и вызовет ошибку.
- **Не меняйте ту же таблицу** *(без **FOR UPDATE**)*. Пытаться **изменить таблицу**, по которой вы в данный момент итерируетесь – **опасно** и может **привести к ошибкам** или **«зависшим» данным**.
- **Избегайте «медленных» операций.** Не помещайте в цикл **очень долгие функции** *(например, запросы к внешним API)*. Обработка ***«строка за строкой»*** может **заблокировать таблицу** **надолго** и **сильно** **замедлить всю систему**.

Спасибо за внимание