ПРАКТИЧЕСКАЯ РАБОТА №6. ТРИГГЕРЫ И КУРСОРЫ В POSTGRESQL

Цель работы:

Целью данной практической работы является формирование у студентов углубленных практических навыков по управлению данными и реализации сложной бизнес-логики в СУБД PostgreSQL с использованием триггеров и курсоров.

По завершении работы студент должен уметь:

- Сформировать практический навык реализации сложной бизнес-логики и ограничений целостности, которые невозможно определить декларативными средствами (**CHECK**, **FOREIGN KEY**), с помощью триггерных механизмов PostgreSQL.
- Научиться создавать триггерные функции на языке **PL/pgSQL**, управляя ходом выполнения DML-операций через анализ специальных переменных (**NEW**, **OLD**) и возвращаемые значения.
- Освоить концепцию **курсоров** для итеративной (построчной) обработки результатов SQL-запроса, понимая их жизненный цикл: объявление (**DECLARE**), открытие (**OPEN**), извлечение данных (**FETCH**) и закрытие (**CLOSE**).
- Развить умение работать с метаданными базы данных (системным каталогом **information_schema**) для создания динамических и универсальных скриптов.
- Получить навык написания сложных функций на **PL/pgSQL**, комбинируя циклы, динамическое выполнение SQL (**EXECUTE**) и обработку данных через курсоры.

Постановка задачи:

Для выполнения практической работы необходимо последовательно выполнить два задания, демонстрирующих применение триггеров и курсоров.

Ваша задача — адаптировать каждую из поставленных задач к логической структуре и предметной области вашей базы данных.

Задание №1: триггеры

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

ВАЖНО: триггеры должны быть **разными**.

Для каждого правила:

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

Как выбрать сценарии для триггеров?

Чтобы вам было проще определить бизнес-правила для вашей базы данных, ищите сценарии, которые подпадают под следующие стандартные категории:

1. **Аудит и логирование.**
   Запись факта изменения данных.
   Пример: при любом изменении столбца salary в таблице employees создавать запись в salary_log, сохраняя **OLD**.salary, **NEW**.salary и CURRENT_USER.

2. **Сложная валидация (проверка).**
   Запрет операции на основе данных из другой таблицы.
   Пример: триггер **BEFORE INSERT** на таблицу order_items должен проверить, что **NEW**.quantity не превышает quantity_on_hand в таблице products. Если превышает – вызвать **RAISE EXCEPTION**.

3. **Поддержание согласованности (денормализация).**
   Автоматическое обновление связанных данных.
   Пример: триггер **AFTER INSERT** на таблицу sales автоматически обновляет столбец total_spent в таблице customers.

4. **Защита данных.**
   Запрет определенных операций.
   Пример: триггер **BEFORE DELETE** на таблицу departments может запретить удаление, если в таблице employees еще есть сотрудники, ссылающиеся на этот отдел.

Задание №2: курсоры

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

- С использованием **явного** **курсора** (**DECLARE/OPEN/FETCH/CLOSE**).
- С использованием **неявного** **курсора** (цикл **FOR...IN**).

Каждый SQL-запрос **сопроводить комментарием**, объясняющим его назначение и логику работы с учетом специфики вашей базы данных.

Подготовка базы данных:

Если ваша база данных не содержит достаточно таблиц и/или полей для реализации трёх полезных триггеров, либо одного или нескольких курсоров, необходимо **дополнить** её, добавив необходимые поля и таблицы.

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

ХОД ВЫПОЛНЕНИЯ РАБОТЫ

Введение

Для создания понятных и последовательных примеров в рамках данной работы будет использоваться упрощенная схема базы данных «Аптека».

Эта схема послужит основой для демонстрации работы триггеров и курсоров. Студентам следует адаптировать представленные примеры для своих собственных баз данных.

Ниже приводятся таблицы, используемые для построения запросов.

Таблица 1. Таблица manufacturers (Производители)

	manufacturer_id	manufacturer_name	country
1	1	ООО "Фармстандарт"	Россия
2	2	Bayer AG	Германия

Таблица 2. Таблица medicines (Лекарства)

	id	name	quantity_in_stock	price	production_date	expiration_date	manufacturer_id
1	1	Парацетамол	200	50,5	2025-07-10	2028-07-10	1
2	2	Аспирин	150	120	2025-07-12	2027-07-12	2
3	3	Ибупрофен	100	85	2025-07-11	2028-07-11	1
4	4	Витамин С	300	250	2025-07-10	2027-07-10	2
5	5	Активированный уголь	500	25	2025-07-10	2030-07-10	1
6	8	Корвалол	140	45	2025-06-01	2029-06-01	1

Таблица 3. Таблица sale_items (Проданные товары)

	sale_item_id	sale_id	medicine_id	quantity	unit_price
1	1	1	1	2	50,5
2	2	1	2	1	120
3	3	2	3	3	85
4	4	2	1	1	50,5
5	5	2	4	5	250

Таблица 4. Таблица customer_audit (Аудит)

	user_id	first_name	last_name	phone_number	change_date
1	1	Иван	Иванов	+79001234567	2025-10-20 01:26:02.183

Таблица 5. Таблица customers (Клиенты)

	customer_id	first_name	last_name	email	phone_number
1	2	Петр	Петров	petr@example.com	[NULL]
2	1	Иван	Иванов	ivan@example.com	+79998887766

Таблица 6. Таблица sales (Продажи)

	sale_id	customer_id	sale_date	total_amount
1	1	1	2025-01-22	221
2	2	[NULL]	2025-01-25	1 555,5

1. РЕАЛИЗАЦИЯ СЛОЖНОЙ БИЗНЕС-ЛОГИКИ С ПОМОЩЬЮ ТРИГГЕРОВ

Обратите внимание: в некоторых СУБД (например, в **MySQL**) допускается написание исполняемого кода триггера внутри самого триггера.

Однако, в **PostgreSQL** это недопустимо, т.к. принято разделение ответственности:

- Триггер (`CREATE TRIGGER`) – это только объявление события (когда, где и при каком условии что-то запускать).
- Триггерная функция (`CREATE FUNCTION ... RETURNS TRIGGER`) – это исполняемая логика (что именно делать, когда событие произошло).

Такой подход обеспечивает модульность и позволяет многократно использовать одну и ту же функцию для разных триггеров.

1.1 Анализ ограничений целостности: когда декларативных правил недостаточно?

Стандартные (декларативные) ограничения целостности, такие как **PRIMARY KEY**, **FOREIGN KEY**, **UNIQUE**, **NOT NULL** и **CHECK**, являются мощным инструментом для поддержания согласованности данных.

Однако их возможности ограничены: они могут проверять значения только в пределах одной строки или ссылаться на первичный ключ другой таблицы. Они не могут выполнять проверки, которые требуют обращения к другим таблицам (кроме **FOREIGN KEY**), производить сложные вычисления или выполнять побочные действия, такие как запись в журнал.

Триггеры преодолевают эти ограничения, позволяя выполнить произвольный код в ответ на DML-операции. Рассмотрим бизнес-правила для «Аптеки», которые невозможно реализовать без триггеров:

1. **Межтабличная проверка**: «запретить продажу товара (добавление записи в sale_items), если на складе (medicines) недостаточное количество

товара (quantity_in_stock)».
Ограничение **CHECK** не может «видеть» данные в другой таблице.

2. **Аудит/логирование**: «при каждом изменении данных (например, phone_number) в таблице customers, необходимо сохранять старое значение этого поля в отдельную таблицу аудита customer_audit».

3. **Модификация связанных данных**: «при добавлении новой продажи в sale_items, необходимо автоматически уменьшать quantity_in_stock в medicines **И** пересчитывать total_amount в sales».

1.2 Триггер 1: сложная валидация

1.2.1 Описание алгоритма

Прежде чем писать код, необходимо четко определить алгоритм работы триггера.

Нам необходимо запретить добавление товара в чек (sale_items), если его недостаточно на складе (medicines.quantity_in_stock). Если товар в наличии, необходимо принудительно установить актуальную цену (medicines.price) в запись о продаже (sale_items.unit_price), чтобы избежать продажи по неверной (старой) цене.

Таблица: **sale_items** *(позиции продаж)*.

Событие: **INSERT** *(добавление товара в чек)*.

Время срабатывания: **BEFORE** *(необходимо проверить остатки **до того**, как запись о продаже будет создана)*.

Уровень: **FOR EACH ROW** *(проверяем каждую позицию товара)*.

Логика действий:

1. Во время **INSERT** в ***sale_items*** получить ***quantity_in_stock*** и ***price*** из таблицы ***medicines*** для ***NEW.medicine_id***.

2. Важно заблокировать (**FOR UPDATE**) строку в ***medicines*** на время проверки, чтобы предотвратить «состояние гонки» (когда два покупателя одновременно пытаются купить последний товар).

3. Если ***stock_quantity*** < ***NEW.quantity*** *(товара не хватает)*, **прервать** операцию с ошибкой (**RAISE EXCEPTION**).

4. **Принудительно** записать актуальную цену в ***NEW.unit_price***.

5. Вернуть **NEW** (разрешить **INSERT**).

1.2.2 Код триггерной функции и триггера

Преобразуем наш алгоритм в код на **PL/pgSQL**.

В начале создаётся функция, которая будет содержать логику, а затем — сам триггер, который связывает эту функцию с таблицей и событием.

Листинг 1. Функция валидации, выполняемая ПЕРЕД вставкой в sale_items

```
CREATE OR REPLACE FUNCTION validate_sale_item_insert()
RETURNS TRIGGER AS $$
DECLARE
  stock_quantity INTEGER;
  med_price      DECIMAL(10, 2);
BEGIN
  SELECT quantity_in_stock, price
  INTO stock_quantity, med_price
  FROM medicines
  WHERE id = NEW.medicine_id
  FOR UPDATE; -- Защита от "состояния гонки"

  IF stock_quantity < NEW.quantity THEN
     RAISE EXCEPTION 'Недостаточно лекарства на складе (ID: %).
Доступно: %, требуется: %',
                     NEW.medicine_id, stock_quantity,
NEW.quantity;
  END IF;

  NEW.unit_price := med_price;

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

В начале мы получаем остаток и цену, блокируя строку от параллельных изменений.

Затем проверяем остаток на складе.

После этого устанавливаем актуальную цену из таблицы лекарств.

И наконец, разрешаем вставку, возвращая изменённую строку.

Статистика 1

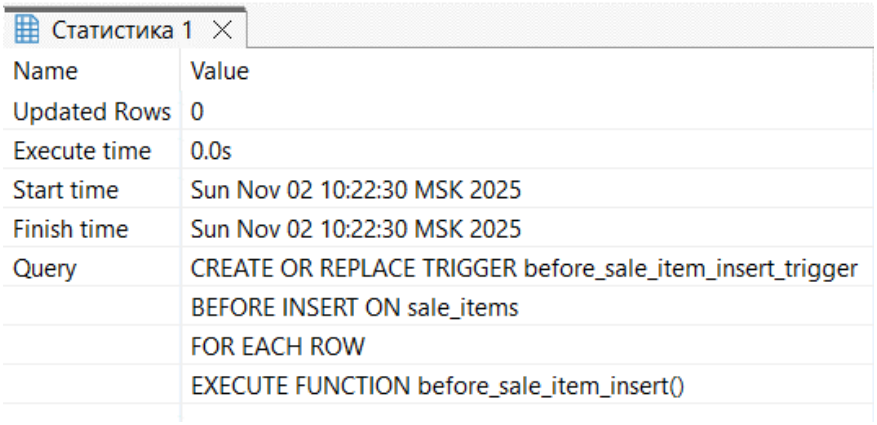
Name	Value
Updated Rows	0
Execute time	0.0s
Start time	Sun Nov 02 10:21:20 MSK 2025
Finish time	Sun Nov 02 10:21:20 MSK 2025
Query	CREATE OR REPLACE FUNCTION validate_sale_item_insert()
	RETURNS TRIGGER AS $$
	DECLARE
	stock_quantity INTEGER;
	med_price DECIMAL(10, 2);
	BEGIN
	SELECT quantity_in_stock, price
	INTO stock_quantity, med_price
	FROM medicines
	WHERE id = NEW.medicine_id
	FOR UPDATE; -- Защита от "состояния гонки"
	IF stock_quantity < NEW.quantity THEN
	RAISE EXCEPTION 'Недостаточно лекарства на складе (ID: %). Доступно: %, требуется: %',
	NEW.medicine_id, stock_quantity, NEW.quantity;
	END IF;
	NEW.unit_price := med_price;
	RETURN NEW;
	END;
	$$ LANGUAGE plpgsql

Рисунок 1 – Результат создания 1 триггерной функции

Далее нам необходимо привязать созданную нами триггерную функцию к конкретному событию, когда она будет срабатывать. Задаём все параметры, которые обговорили на этапе описания алгоритма.

Листинг 2. Привязка триггера BEFORE INSERT

```
CREATE OR REPLACE TRIGGER before_sale_item_insert_trigger
BEFORE INSERT ON sale_items
FOR EACH ROW
EXECUTE FUNCTION before_sale_item_insert();
```


Статистика 1

Name	Value
Updated Rows	0
Execute time	0.0s
Start time	Sun Nov 02 10:22:30 MSK 2025
Finish time	Sun Nov 02 10:22:30 MSK 2025
Query	CREATE OR REPLACE TRIGGER before_sale_item_insert_trigger
	BEFORE INSERT ON sale_items
	FOR EACH ROW
	EXECUTE FUNCTION before_sale_item_insert()

Рисунок 2 – Результат привязки 1 триггерной функции

1.2.3 Демонстрация (тестирование) работы триггера

Для теста используем данные «Парацетамола» (id: 1, на складе: 200, цена: 50.5).

ТЕСТ 1: попытка продать 300 единиц, хотя доступно только 200.

Ожидаемый результат: провал, так как товара на складе недостаточно.

Листинг 3. Запрос, который должен быть прерван триггером

```
INSERT INTO sale_items (sale_id, medicine_id, quantity)
VALUES (1, 1, 300);
```

Полученный результат полностью совпадает с ожидаемым – Рисунок 3.

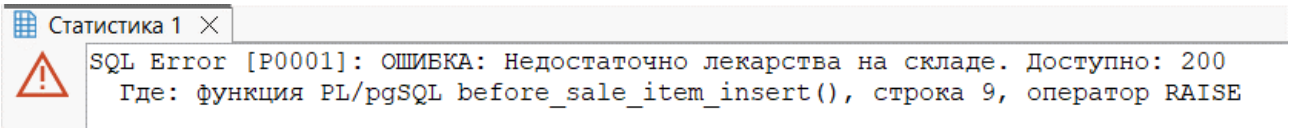


Рисунок 3 – Результаты проверки

ТЕСТ 2: попытка продать 10 единиц (доступно 200). Мы также не указываем ***unit_price***, ожидая, что триггер подставит верное значение (50.5).

Ожидаемый результат: успех, так как товара на складе достаточно.

Листинг 4. Запрос, который должен выполниться успешно

```
INSERT INTO sale_items (sale_id, medicine_id, quantity)
VALUES (1, 1, 10);
```

Полученный результат снова полностью совпадает с ожидаемым, как видно по **Updated Rows**, которое говорит о том, сколько было изменено строк – Рисунок 4.

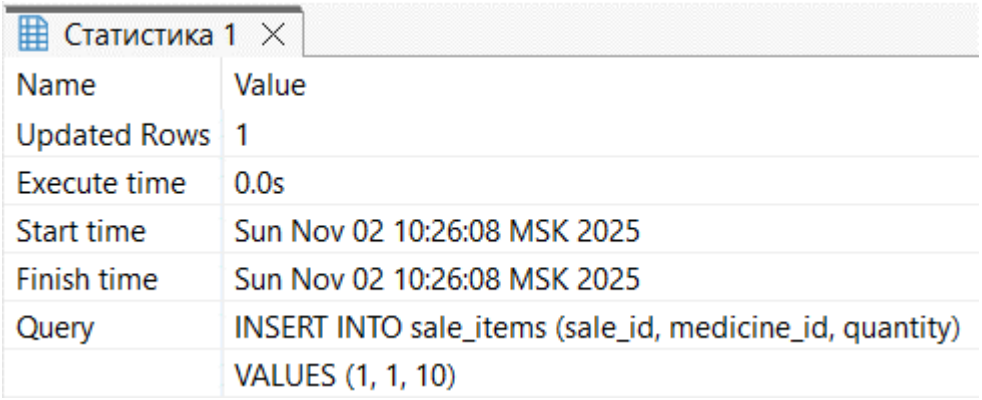


Рисунок 4 – Результаты проверки

Триггер был успешно реализован и прошёл необходимые проверки.

1.3 Триггер 2: поддержание согласованности

Наш **первый триггер** осуществлял **валидацию** – как охранник, который стоит **ПЕРЕД** (**BEFORE**) операцией и решает, можно ли её в принципе выполнить (хватит ли товара?).

Второй триггер – это «бухгалтер-кладовщик».

Ему всё равно, можно или нельзя – он знает, что операция **УЖЕ** произошла (**AFTER**), и его задача – просто **навести порядок в учёте**.

1.3.1 Описание алгоритма

Сформулируем задачу для 2 триггера.

После любой операции (**INSERT**, **UPDATE**, **DELETE**) с ***sale_items***, система должна:

1. Скорректировать остаток (***quantity_in_stock***) в ***medicines*** *(**списать** при **INSERT**, **вернуть** при **DELETE**, **скорректировать** при **UPDATE**).*
2. Пересчитать итоговую сумму (***total_amount***) в чеке (***sales***).

Таблица: **sale_items** *(позиции продаж).*

Событие: **INSERT OR UPDATE OR DELETE** *(добавление товара в чек).*

Время срабатывания: **AFTER** *(необходимо проверить остатки **после того**, как запись о продаже будет создана).*

Уровень: **FOR EACH ROW** *(проверяем каждую позицию товара).*

Логика действий:

1. Определить ***target_sale_id*** (из **NEW** или **OLD**).
2. В зависимости от **TG_OP** (тип операции):
   1.1. **INSERT**: списать **NEW.quantity** со склада (**UPDATE *medicines***...).
   1.2. **DELETE**: вернуть **OLD.quantity** на склад (**UPDATE *medicines***...).
   1.3. **UPDATE**: скорректировать склад на разницу (**NEW.quantity** - **OLD.quantity**).
3. Пересчитать **SUM(quantity * unit_price)** для ***target_sale_id*** в ***sale_items***.
4. Обновить ***total_amount*** в ***sales*** этой суммой.

1.3.2 Код триггерной функции и триггера

Преобразуем наш алгоритм в код на **PL/pgSQL**.

В начале создаётся функция, которая будет содержать логику, а затем — сам триггер, который связывает эту функцию с таблицей и событием.

Листинг 5. Функция, выполняемая ПОСЛЕ вставки/обновления/удаления

```
CREATE OR REPLACE FUNCTION update_aggregates_after_sale_change()
RETURNS TRIGGER AS $$
DECLARE
    sale_total      DECIMAL(10, 2);
    target_sale_id  INT;
BEGIN
    IF (TG_OP = 'DELETE') THEN
        target_sale_id := OLD.sale_id;
        UPDATE medicines
        SET quantity_in_stock = quantity_in_stock + OLD.quantity
        WHERE id = OLD.medicine_id;
    ELSE
        target_sale_id := NEW.sale_id;
        IF (TG_OP = 'INSERT') THEN
            UPDATE medicines
            SET quantity_in_stock =
                quantity_in_stock - NEW.quantity
            WHERE id = NEW.medicine_id;
        ELSIF (TG_OP = 'UPDATE') THEN
            UPDATE medicines
            SET quantity_in_stock =
                quantity_in_stock - (NEW.quantity - OLD.quantity)
```

```
            WHERE id = NEW.medicine_id;
        END IF;
    END IF;

    SELECT COALESCE(SUM(quantity * unit_price), 0)
    INTO sale_total
    FROM sale_items
    WHERE sale_id = target_sale_id;

    UPDATE sales
    SET total_amount = sale_total
    WHERE sale_id = target_sale_id;

    IF (TG_OP = 'DELETE') THEN
        RETURN OLD;
    ELSE
        RETURN NEW;
    END IF;
END;
$$ LANGUAGE plpgsql;
```

Итак, здесь мы:

1. Определяем ID чека и ***корректируем склад***, причём:
   1.1. При **DELETE** - возвращаем товар на склад.
   1.2. При **INSERT** - списываем товар (1 триггер уже проверил наличие)
   1.3. При **UPDATE** - корректируем на разницу (**учтите**, что в **реальном проекте** здесь также **должна быть проверка**, похожая на ту, что была реализована в **1 триггере** - ведь мы проверяли только **INSERT**, **не UPDATE**!
   Здесь она ***опущена для упрощения***, но в реальном проекте такое **недопустимо**!)
2. Пересчитываем итоговую сумму чека
3. Обновляем сумму в таблице 'sales'
4. Возвращаемое значение для **AFTER**-триггера **игнорируется**, но **мы его заполняем**, так как в PostgreSQL **любая функция**, которая должна быть вызвана через `EXECUTE FUNCTION` в `CREATE TRIGGER`, синтаксически **обязана** быть объявлена с возвращаемым типом `TRIGGER`.

При этом в PostgreSQL из функции, объявленной как **RETURNS TRIGGER**, разрешено возвращать **только** **три вида значений**:

- Запись NEW
- Запись OLD
- NULL

Статистика 1

Name	Value
Updated Rows	0
Execute time	0.0s
Start time	Sun Nov 02 10:54:02 MSK 2025
Finish time	Sun Nov 02 10:54:02 MSK 2025
Query	CREATE OR REPLACE FUNCTION update_aggregates_after_sale_change()
	RETURNS TRIGGER AS $$
	DECLARE
	sale_total DECIMAL(10, 2);
	target_sale_id INT;
	BEGIN
	IF (TG_OP = 'DELETE') THEN
	target_sale_id := OLD.sale_id;
	UPDATE medicines
	SET quantity_in_stock = quantity_in_stock + OLD.quantity
	WHERE id = OLD.medicine_id;
	ELSE
	target_sale_id := NEW.sale_id;
	IF (TG_OP = 'INSERT') THEN
	UPDATE medicines
	SET quantity_in_stock = quantity_in_stock - NEW.quantity
	WHERE id = NEW.medicine_id;
	ELSIF (TG_OP = 'UPDATE') THEN
	UPDATE medicines
	SET quantity_in_stock = quantity_in_stock - (NEW.quantity - OLD.quantity)
	WHERE id = NEW.medicine_id;
	END IF;
	END IF;
	SELECT COALESCE(SUM(quantity * unit_price), 0)
	INTO sale_total
	FROM sale_items
	WHERE sale_id = target_sale_id;
	UPDATE sales
	SET total_amount = sale_total
	WHERE sale_id = target_sale_id;
	IF (TG_OP = 'DELETE') THEN
	RETURN OLD;
	ELSE
	RETURN NEW;
	END IF;
	END;
	$$ LANGUAGE plpgsql

Рисунок 5 – Результат создания 2 триггерной функции

Далее нам снова необходимо привязать созданную нами триггерную функцию к конкретному событию, когда она будет срабатывать. Задаём все параметры, которые обговорили на этапе описания алгоритма.

Листинг 6. Привязка триггера AFTER

```
CREATE OR REPLACE TRIGGER trg_update_aggregates_after_sale
AFTER INSERT OR UPDATE OR DELETE ON sale_items
FOR EACH ROW
EXECUTE FUNCTION update_aggregates_after_sale_change();
```

Статистика 1

Name	Value
Updated Rows	0
Execute time	0.0s
Start time	Sun Nov 02 11:02:07 MSK 2025
Finish time	Sun Nov 02 11:02:07 MSK 2025
Query	CREATE OR REPLACE TRIGGER trg_update_aggregates_after_sale
	AFTER INSERT OR UPDATE OR DELETE ON sale_items
	FOR EACH ROW
	EXECUTE FUNCTION update_aggregates_after_sale_change()

Рисунок 6 – Результат привязки 2 триггерной функции

1.3.3 Демонстрация (тестирование) работы триггера

Этот триггер срабатывает ПОСЛЕ (AFTER) того, как вы добавили, удалили или изменили строку в sale_items (в корзине/чеке).

У него две независимые задачи:

1. Задача **Кладовщика** *(обновить склад)*:

   - Если был **INSERT** (добавили товар в чек) – он идёт в medicines и уменьшает quantity_in_stock.
   - Если был **DELETE** (удалили товар из чека) – он идёт в medicines и увеличивает (возвращает) quantity_in_stock.
   - Если был **UPDATE** (изменили кол-во с 2 на 5) – он корректирует quantity_in_stock на разницу (в данном случае, спишет ещё 3).

2. Задача **Бухгалтера** *(обновить сумму чека)*:

- После **любого** из этих действий (**INSERT**, **DELETE**, **UPDATE**) сумма чека в таблице ***sales*** становится неправильной.
- Триггер заново полностью **пересчитывает** (**SUM**) **все товары** в этом чеке и обновляет ***total_amount*** в ***sales***.

Давайте проверим, как он выполняет эти задачи. Для тестов используем данные «Парацетамола».

Тест 1: попробуем продать 10 единиц товара.

Состояние «до» (для **INSERT**):

- Парацетамол (id: 1): ***quantity_in_stock*** = 200.
- Чек (id: 1): ***total_amount*** = 221

Ожидаемый результат:

- Задача **Кладовщика**: **списать** 10 шт. со склада.
  (Остаток должен стать 200 - 10 = **190**).
- Задача **Бухгалтера**: **пересчитать** сумму чека.
  (Сумма должна стать 221 + (10 * 50.5) = **726**).

Листинг 7. Активация триггеров (INSERT)

```
INSERT INTO sale_items (sale_id, medicine_id, quantity)
VALUES (1, 1, 10);

SELECT quantity_in_stock FROM medicines WHERE id = 1;
SELECT total_amount FROM sales WHERE sale_id = 1;
```

Полученный результат полностью совпадает с ожидаемым – Рисунок 3.

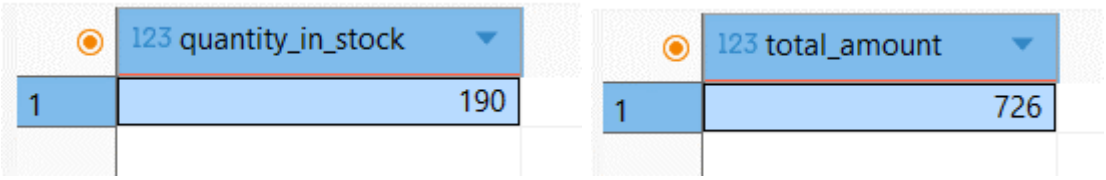


Рисунок 7 – Результаты проверки

ТЕСТ 2: выполняем **DELETE**, удаляя 10 шт. «Парацетамола» из чека.

Состояние «до» (для **UPDATE**):

- Парацетамол (id: 1): ***quantity_in_stock*** = 190.
- Чек (id: 1): ***total_amount*** = 726

Ожидаемый результат:

- Задача **Кладовщика**: **вернуть** 10 шт. на склад.
  (Остаток должен стать 190 + 10 = **200**).
- Задача **Бухгалтера**: **пересчитать** сумму чека.
  (Сумма должна стать 726 – (10 * 50.5) = **221**).

Листинг 8. Запрос, который должен выполниться успешно

```
DELETE FROM sale_items
WHERE sale_id = 1 AND medicine_id = 1 AND quantity = 10;

SELECT quantity_in_stock FROM medicines WHERE id = 1;
SELECT total_amount FROM sales WHERE sale_id = 1;
```

Полученный результат снова полностью совпадает с ожидаемым, тест успешно пройден.

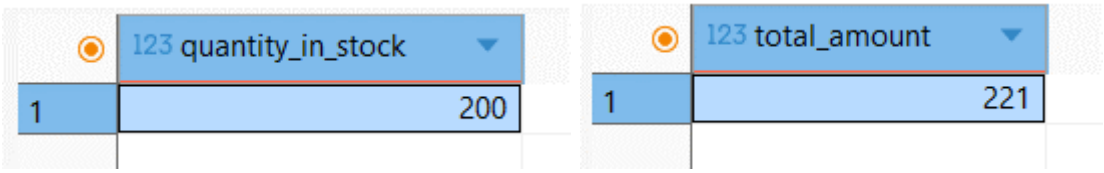


Рисунок 8 – Результаты проверки

1.4 Триггер 3: аудит и логирование

1.4.1 Описание алгоритма

Прежде чем писать код, необходимо четко определить алгоритм работы триггера.

При изменении только номера телефона (***phone_number***) в таблице ***customers***, необходимо сохранить старые данные (**ФИО**, **старый номер**) в таблицу аудита ***customer_audit***.

Таблица: **customers** *(клиенты)*.

Событие: **UPDATE** *(обновление номера телефона)*.

Время срабатывания: **AFTER** *(необходимо залогировать действия **после того**, как они произошли)*.

Уровень: **FOR EACH ROW** *(проверяем каждую позицию)*.

Логика действий:

1. Проверить, изменился ли ***phone_number*** (`IF NEW.phone_number IS DISTINCT FROM OLD.phone_number`).
2. Если **да**, выполнить **INSERT** в *customer_audit*, используя данные из **OLD** *(старая версия строки)* и **NOW()** *(текущее время)*.

1.4.2 Код триггерной функции и триггера

Преобразуем наш алгоритм в код на **PL/pgSQL**.

В начале создаётся функция, которая будет содержать логику, а затем — сам триггер, который связывает эту функцию с таблицей и событием.

Листинг 9. Функция логирования изменений телефона

```
CREATE OR REPLACE FUNCTION log_customer_phone_change()
RETURNS TRIGGER AS $$
BEGIN
  IF NEW.phone_number IS DISTINCT FROM OLD.phone_number THEN
    INSERT INTO customer_audit
      (user_id, first_name, last_name, phone_number, change_date)
    VALUES
      (OLD.customer_id, OLD.first_name, OLD.last_name,
OLD.phone_number, NOW());
  END IF;

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

Делаем проверку - триггер срабатывает, только если **номер телефона ДЕЙСТВИТЕЛЬНО изменился**.

Как говорилось ранее, для **AFTER**-триггера **RETURN** не важен, но **синтаксически нужен**.

Статистика 1

Name	Value
Updated Rows	0
Execute time	0.0s
Start time	Sun Nov 02 11:30:12 MSK 2025
Finish time	Sun Nov 02 11:30:12 MSK 2025
Query	CREATE OR REPLACE FUNCTION log_customer_phone_change()
	RETURNS TRIGGER AS $$
	BEGIN
	IF NEW.phone_number IS DISTINCT FROM OLD.phone_number THEN
	INSERT INTO customer_audit
	(user_id, first_name, last_name, phone_number, change_date)
	VALUES
	(OLD.customer_id, OLD.first_name, OLD.last_name, OLD.phone_number, NOW());
	END IF;
	RETURN NEW;
	END;
	$$ LANGUAGE plpgsql

Рисунок 9 – Результат создания 3 триггерной функции

Далее нам необходимо привязать созданную нами триггерную функцию к конкретному событию, когда она будет срабатывать. Задаём все параметры, которые обговорили на этапе описания алгоритма.

Листинг 10. Привязка триггера AFTER UPDATE

```
CREATE OR REPLACE TRIGGER trg_log_customer_phone_change
AFTER UPDATE ON customers
FOR EACH ROW
EXECUTE FUNCTION log_customer_phone_change();
```

Статистика 1

Name	Value
Updated Rows	0
Execute time	0.0s
Start time	Sun Nov 02 11:31:02 MSK 2025
Finish time	Sun Nov 02 11:31:02 MSK 2025
Query	CREATE OR REPLACE TRIGGER trg_log_customer_phone_change
	AFTER UPDATE ON customers
	FOR EACH ROW
	EXECUTE FUNCTION log_customer_phone_change()

Рисунок 10 – Результат привязки 1 триггерной функции

1.4.3 Демонстрация (тестирование) работы триггера

Состояние «до»:

- В customers есть "Иван Иванов" (id: 1) с номером +79998887766.
- В customer_audit есть 1 запись.

ТЕСТ 1: меняем email клиента.

Ожидаемый результат: триггер **не должен** сработать.

Листинг 11. Запрос, который должен быть прерван триггером

```
UPDATE customers
SET email = 'ivan_new@example.com'
WHERE customer_id = 1;

SELECT COUNT(*) FROM customer_audit;
```

Полученный результат полностью совпадает с ожидаемым. Количество строк не изменилось.

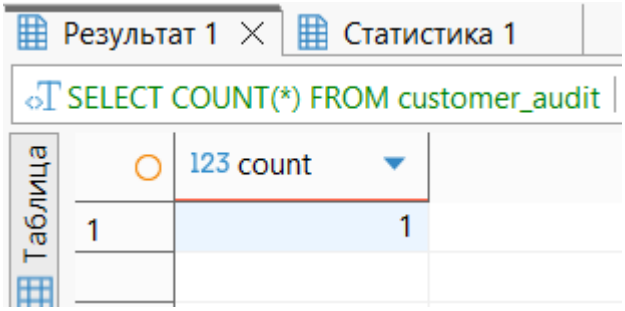


Рисунок 11 – Результаты проверки

ТЕСТ 2: меняем номер телефона клиента.

Ожидаемый результат: триггер **должен** сработать.

Листинг 12. Запрос, который должен выполниться успешно

```
UPDATE customers
SET phone_number = '+71112223344'
WHERE customer_id = 1;

SELECT COUNT(*) FROM customer_audit;
```

Полученный результат снова полностью совпадает с ожидаемым, теперь количество записей равно 2.

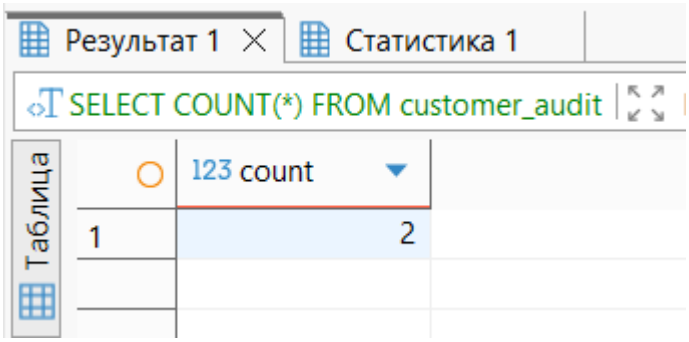


Рисунок 12 – Результаты проверки

Триггер был успешно реализован и прошёл необходимые проверки.

2. ИСПОЛЬЗОВАНИЕ КУРСОРОВ

2.1 Основы работы с курсорами: зачем нужен построчный доступ?

Стандартные SQL-запросы являются декларативными и ориентированы на работу с множествами данных. Однако иногда возникают задачи, требующие процедурного подхода — выполнения действий для каждой строки результата по очереди.

Курсор — это объект базы данных, который инкапсулирует результирующий набор запроса и позволяет перемещаться по нему строка за строкой, как будто это цикл в языке программирования.

В **PL/pgSQL** есть два основных способа работы с ними:

- явный (ручной)
- неявный (автоматический).

2.2 Явный курсор (DECLARE / OPEN / FETCH / CLOSE)

Этот способ дает полный контроль над процессом. Мы вручную объявляем, открываем, получаем данные в цикле и закрываем курсор.

Алгоритм:

1. **Объявление** ***(DECLARE)*** – связать курсор **curs1** с запросом:
   `SELECT id, name FROM medicines`.
2. **Открытие** ***(OPEN)*** – выполнить запрос и подготовить курсор к чтению.
3. **Цикл** ***(LOOP)***:
   - **Извлечение** ***(FETCH)*** – получить следующую строку из курсора в переменную **row_var**.
   - **Проверка** ***(EXIT WHEN NOT FOUND)*** – если **FETCH** не вернул строку *(данные кончились)*, выйти из цикла.

- **Обработка** ***(RAISE NOTICE)*** – вывести содержимое **row_var** в консоль.

4. **Закрытие** ***(CLOSE)*** – освободить ресурсы, связанные с курсором.

Листинг 13. Пример явного курсора

```
DO $$
DECLARE
  curs1 CURSOR FOR SELECT id, name FROM medicines;
  row_var RECORD;
BEGIN
  OPEN curs1;
  LOOP
    FETCH curs1 INTO row_var;
    EXIT WHEN NOT FOUND;
    RAISE NOTICE 'ID: %, Название: %', row_var.id, row_var.name;
  END LOOP;
  CLOSE curs1;
END;
$$ LANGUAGE plpgsql;
```

Реализуем вышеописанный алгоритм, и смотрим на результат (он будет выведен в консоли).

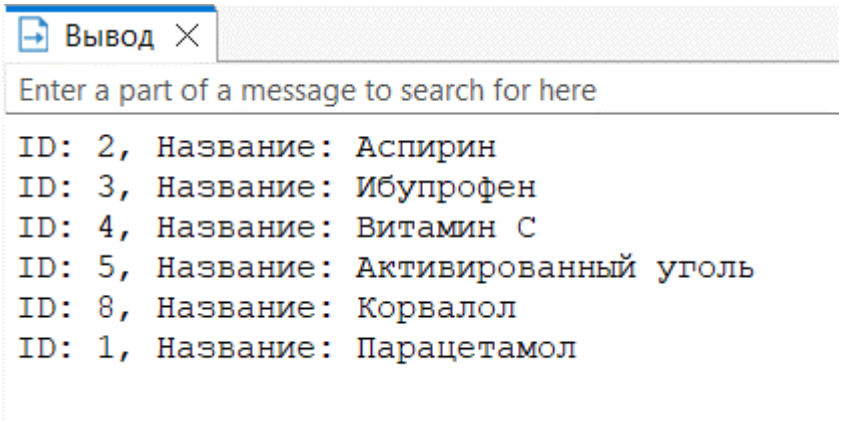


Рисунок 13 – Результат запроса

2.3 Неявный курсор в функции – вывод результата в таблицу

Неявный курсор – это предпочтительный, более чистый и безопасный способ для большинства задач. **PL/pgSQL** автоматически управляет всем жизненным циклом курсора ***(DECLARE, OPEN, FETCH, CLOSE)***.

Ранее мы всегда выводили результаты в консоль, что, по очевидным причинам, не слишком применимо в реальных проектах. На самом деле, чтобы вернуть результат в виде таблицы, мы просто должны создать **функцию**, которая возвращает **RETURNS TABLE(<поля>)**.

Внутри этой функции мы используем цикл **FOR...IN** *(неявный курсор)* и команду **RETURN NEXT**, чтобы добавить каждую обработанную строку в **итоговый табличный результат**.

Алгоритм:

1. **Создание функции** ***(CREATE FUNCTION)*** – определить функцию, возвращающую таблицу:
   `RETURNS TABLE(customer_id_out INT, full_name TEXT, email_out VARCHAR(255))`.
2. **Объявление** ***(DECLARE)*** – определить переменную-приемник customer_row (например, с типом RECORD или %ROWTYPE).
3. **Цикл** ***(FOR...IN)*** – запустить цикл по запросу `SELECT * FROM customers ORDER BY customer_id`.
   PostgreSQL автоматически выполнит шаги (**OPEN**, **FETCH** в ***customer_row*** и **CLOSE**).
4. **Обработка и возврат** ***(RETURN NEXT)*** – внутри цикла присвоить значения выходным столбцам и вызвать `RETURN NEXT`, чтобы добавить строку в результат.
5. **Вызов функции** – получить итоговую таблицу с помощью `SELECT * FROM название_функции();`

Листинг 14. Создание функции, возвращающей таблицу (с неявным курсором)

```
CREATE OR REPLACE FUNCTION get_all_customers_formatted()
RETURNS TABLE(customer_id_out INT, full_name TEXT, email_out
VARCHAR(255))
AS $$
DECLARE
    customer_row customers%ROWTYPE;
BEGIN
    FOR customer_row IN
        SELECT * FROM customers ORDER BY customer_id
    LOOP
        customer_id_out := customer_row.customer_id;
        full_name := customer_row.first_name || ' ' ||
customer_row.last_name;
        email_out := customer_row.email;

        RETURN NEXT;
    END LOOP;

    RETURN;
END;
$$ LANGUAGE plpgsql;

SELECT * FROM get_all_customers_formatted();
```

Ожидаемый результат: в отличие от Листинга 10, этот **SELECT** должен вернуть полноценную таблицу (во вкладке «Data» / «Result»).

	123 customer_id_out	A-Z full_name	A-Z email_out
1	1	Иван Иванов	ivan@example.com
2	2	Петр Петров	petr@example.com

Рисунок 14 – Результат запроса

И как мы видим, именно этот результат мы и получаем. Теперь мы сможем использовать этот результат либо напрямую, либо в других запросах.

КОНТРОЛЬНЫЕ ВОПРОСЫ

1. В чем принципиальное отличие триггера уровня **STATEMENT** от триггера уровня **ROW**?
   Приведите пример бизнес-задачи для вашей предметной области, где был бы уместен именно **STATEMENT**-level триггер.

2. Что произойдет с DML-операцией ***(INSERT** или **UPDATE)***, если триггерная функция, вызванная в режиме **BEFORE EACH ROW**, вернет значение **NULL**?
   А если она вернет **OLD** в ответ на операцию **UPDATE**?

3. В Задании 2 мы использовали **RAISE NOTICE** для вывода в консоль и **RETURN NEXT** *(внутри функции)* для возврата таблицы. Объясните разницу между этими двумя командами.
   Почему **RAISE NOTICE** нельзя использовать для получения таблицы?

4. Какие значения может принимать переменная **TG_OP** и в каких ситуациях они используются?

5. В Триггере 1 (Листинг 1) для отмены операции при нехватке товара используется **RAISE EXCEPTION**.
   Что бы произошло, если бы мы вместо этого использовали **RETURN NULL**?
   В чём разница для пользователя?

КРАТКИЙ СПРАВОЧНЫЙ МАТЕРИАЛ

1. Триггеры: автоматизация реакции на события

1.1 Что такое триггер?

Триггер — это процедура, которая автоматически запускается в ответ на определенное **DML**-событие ***(INSERT, UPDATE или DELETE)*** в таблице, к которой он привязан.

DML – Data Manipulation Language – Язык Манипуляции Данными.

Ключевые компоненты триггера:

Событие *(Event)* – условие активации триггера:

- **INSERT** – вставка данных;
- **UPDATE –** обновление данных;
- **DELETE** – удаление данных;
- **TRUNCATE** – очистка таблицы.

Время *(Timing)* – когда он срабатывает?

- **BEFORE**: до выполнения операции. Позволяет проверить или изменить данные перед их записью.
- **AFTER**: после выполнения операции. Используется для действий, которые должны произойти после успешного изменения данных (например, логирование).
- **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 количества измененных строк.

1.2 Когда использовать BEFORE, когда – AFTER?

Используйте **BEFORE** для:

Валидации – **проверки** данных до того, как они попадут в таблицу. Если **данные неверны**, вы можете вызвать `RAISE EXCEPTION`, чтобы **отменить** операцию.

Модификации данных – **изменения** вставляемых данных «на лету» – до того, как они попадут в таблицу.

Например, привести email к нижнему регистру `NEW.email = LOWER(NEW.email);`

Используйте **AFTER** для:

Логирования *(Аудита)* – записи в журнал аудита. Вы хотите логировать только те операции, которые **уже успешно произошли**.

Обновления связанных таблиц – изменения данных в других таблицах, которые **зависят от успешного завершения** текущей операции *(как в нашем примере с обновлением остатков на складе)*.

1.3 «Волшебные» переменные NEW и OLD

В триггерах уровня **ROW** у вас есть доступ к специальным переменным типа **RECORD**, которые содержат данные строки до и после операции.

Таблица. Переменные NEW и OLD

Операция	OLD *(данные до операции)*	NEW *(данные после операции)*
INSERT	**NULL** *(строки ещё не было)*.	Содержит **вставляемую строку**. **Можно изменять** в **BEFORE**-триггере.
UPDATE	Содержит **старую версию** строки.	Содержит **новую версию** строки. **Можно изменять** в **BEFORE**-триггере.
DELETE	Содержит **удаляемую** строку.	**NULL** *(строки больше не будет)*.

1.4 Роль возвращаемого значения в BEFORE-триггере

То, что ваша триггерная функция возвращает, напрямую влияет на судьбу операции.

RETURN NEW; => «Всё в порядке, продолжай операцию с этой *(возможно, измененной)* строкой».

RETURN NULL; => «Тихо проигнорируй эту строку, не выполняй с ней операцию, но не прерывай транзакцию».

RAISE EXCEPTION; => «Тревога! Останови всё немедленно и откати транзакцию».

2. Курсоры: построчная обработка данных

2.1 Что такое курсор?

Курсор — это «закладка» или указатель, который позволяет вам итеративно *(построчно)* обрабатывать результирующий набор запроса.

Жизненный цикл курсора:

- **DECLARE** *(объявление)* – вы связываете переменную-курсор с SQL-запросом.
- **OPEN** *(открытие)* – выполняется SQL-запрос, и курсор устанавливается в позицию перед первой строкой.
- **FETCH** *(извлечение)* – вы получаете данные из текущей строки в переменные и перемещаете курсор на следующую строку.
- **CLOSE** *(закрытие)* – вы освобождаете ресурсы, связанные с курсором.

Типы курсорных переменных:

- **Связанный** ***(bound)*** – курсор «жёстко» привязан к одному, статически определенному запросу.

```
DECLARE cur CURSOR FOR SELECT * FROM my_table;
```

- **Несвязанный *(Unbound)* / refcursor** – `DECLARE cur refcursor;`

  Это универсальная переменная-указатель. Она не привязана к конкретному запросу при объявлении и может быть открыта для результата любого запроса с помощью `OPEN cur FOR...;`.

  Используется для динамического SQL.

2.2 Альтернатива курсору: цикл FOR...IN

Для простых итераций **PL/pgSQL** предоставляет более удобный синтаксис неявного курсора – цикл `FOR...IN`.

Этот цикл **автоматически** объявляет, открывает, извлекает данные (***fetch***) и закрывает курсор.

Листинг. Пример

```
DECLARE
    row_record RECORD;
BEGIN
    FOR row_record IN SELECT * FROM medicines ORDER BY name
    LOOP
        RAISE NOTICE 'Лекарство: %', row_record.name;
    END LOOP;
END;
$$;
```

Здесь `row_record` **уже содержит** данные **текущей строки**.

3. Ключевые понятия PL/pgSQL

Анонимный блок `DO $$...$$` – способ выполнить блок кода на PL/pgSQL без необходимости создавать постоянную хранимую функцию. Идеально подходит для одноразовых административных скриптов.

Функция `RETURNS TABLE(...)` или `RETURNS SETOF <тип>` – способ создания функции, которая возвращает вызывающей стороне полноценный табличный результат (набор строк), а не одно значение.

RETURN NEXT; – команда, используемая внутри функций, возвращающих TABLE или SETOF. Она добавляет текущую строку в набор результатов, который будет возвращен после завершения функции.

RAISE NOTICE '...'; – команда для вывода информационных сообщений в клиентскую консоль. Основной инструмент для «отладки» и отображения результатов в наших заданиях.

RAISE EXCEPTION '...'; – команда для принудительного прерывания транзакции и возврата ошибки.

TG_OP – специальная переменная, доступная в триггерах. Она содержит текстовую строку, указывающую на операцию, вызвавшую триггер: 'INSERT', 'UPDATE' или 'DELETE'.