

МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ  
ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО  
ОБРАЗОВАНИЯ  
**«НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ ИТМО»**

**Факультет безопасности информационных технологий**

**Направление подготовки: 10.03.01 Информационная безопасность**  
**Образовательная программа: "Информационная безопасность / Information security"**

**Дисциплина:**  
**«*Информационная безопасность баз данных*»**

**ОТЧЕТ ПО ЛАБОРАТОРНОЙ РАБОТЕ № 3**  
**«*Функции и триггеры в БД*»**

**Выполнил студент:**  
группа/поток 1.3  
Бардышев Артём Антонович/  
*Подпись*

**Проверил:**  
Карманова Наталья Андреевна/  
*Подпись*

*Отметка о выполнении (один из вариантов:  
отлично, хорошо, удовлетворительно, зачлено)*

*Дата*

Санкт-Петербург  
2025г.

1. Цель работы: Получение навыков написание процедур, функций и триггеров в БД.
2. Теоретическая информация:

При наличии необходимости реализации сложного сценария работы с данными в БД применяются процедуры и триггеры. Они представляют собой код, написанный на одном из расширений SQL (для PostgreSQL PL/pgSQL), который позволяет реализовать более сложную обработку данных и имеет в своем арсенале операторы циклов, ветвлений и т.п. Процедура является определена функцией, которую может запускать пользователь, используя клиента БД. Триггер представляет собой операцию, которую нужно выполнить при возникновении события в БД, например, вставка записи в таблицу.

В качестве СУБД, используемой в лабораторной работе, предполагается PostgreSQL. В ней триггеры создаются на основе уже определенных ранее функций.

**Оператор для создания процедуры**

```
CREATE OR REPLACE FUNCTION функция() RETURNS тип AS  
$$  
BEGIN  
команды;  
END  
$$ LANGUAGE plpgsql;
```

Для создания хранимой процедуры нужно определить ее в соответствии с представленным синтаксисом описать команды и возвращаемый результат, который может быть простым типом данных, набором данных, сложным набором данных, пользовательским набором данных.

Для последующего вызова хранимой процедуры можно воспользоваться синтаксисом, приведенным ниже:

**Вызов функции**

```
SELECT * FROM функция();
```

**Пример:**

```
CREATE OR REPLACE FUNCTION max_value() RETURNS real AS  
$$  
DECLARE
```

```
maxVal real;  
BEGIN  
    maxVal := (select max(sensor_value) from svalues);  
    RETURN maxVal;  
END  
$$ LANGUAGE plpgsql;
```

Для последующего создания триггера необходимо определить триггерную функцию в соответствии с синтаксисом ниже.

#### Оператор для создания функции для триггера

```
CREATE FUNCTION функция () RETURNS trigger AS $$  
DECLARE  
    объявления;  
BEGIN  
    команды;  
END; $$  
LANGUAGE plpgsql;
```

Пример:

```
CREATE OR REPLACE FUNCTION trigger_update_maxvalue() RETURNS trigger AS  
trigger AS  
$$  
DECLARE  
    maxVal real;  
BEGIN  
    IF NEW.sensor_value > (select current_val from max_value)  
    THEN update max_value set current_val=NEW.sensor_value;  
    END IF;  
    RETURN NEW;  
END;  
$$ LANGUAGE plpgsql;
```

#### Оператор определения триггера

```
CREATE TRIGGER триггер
BEFORE | AFTER } { событие [ OR событие ] } ON таблица
FOR EACH { ROW | STATEMENT }
EXECUTE PROCEDURE функция ( аргументы )
```

**Добавление новых записей в таблицу производится оператором INSERT.**  
Необходимо в явном виде указать таблицу, в которую добавляются строки, а также список атрибутов добавляемой записи и ее значения.

**Пример:**

```
CREATE TRIGGER trigger_keep_maxvalue
AFTER INSERT OR UPDATE ON svalues FOR EACH ROW
EXECUTE PROCEDURE trigger_update_maxvalue();
```

**Все перечисленные операторы имеют возможности по использованию дополнительных параметров, с которыми необходимо ознакомиться в документации.**

### **3. Задание**

- 1. Написать процедуру, которая выполняет агрегацию значений в таблице и обновляет значение в другой таблице. Таким образом, чтобы при запуске пользователем информация в таблице обновлялась и содержала агрегированные значения из другой таблицы.**
- 2. Написать триггер, который будет выполнять действие из 1 пункта автоматически при вставке записи в исходную таблицу. Таким образом, чтобы агрегированная информация всегда была актуальна.**
- 3. Написать триггер, который на основании даты из вставляемой записи, вставлял ее в соответствующую таблицу.**
- 4. Написать триггер, который при вставке в таблицу, производил подмену вставляемого значения в соответствии с уже существующим словарем.**
- 5. Реализуйте триггер, который использует по крайней мере 2-3 специальных переменных (NEW, OLD, TG\_OP и др). Список специальных переменных для postgresql <https://postgrespro.ru/docs/postgresql/9.6/plpgsql-trigger>**

### **4. Требования к оформлению отчета и защите**

**Отчет должен содержать код процедур и триггерных функций, выполненные операторы по каждому пункту задания и вывод, полученный при их выполнении.**

## **5. Источники и информация для подготовки**

<http://www.postgresql.org/docs/9.5/static/index.html>

ХОД РАБОТЫ:

### **1) Подготовка базы данных**

Для выполнения заданий создадим тестовые таблицы:

-- Таблица с исходными значениями

```
CREATE TABLE sensor_data (
    id SERIAL PRIMARY KEY,
    sensor_id INTEGER,
    sensor_value REAL,
    record_date TIMESTAMP
);
```

-- Таблица для агрегированных данных

```
CREATE TABLE aggregated_data (
    sensor_id INTEGER PRIMARY KEY,
    max_value REAL,
    min_value REAL,
    avg_value REAL,
    last_update TIMESTAMP
);
```

-- Таблица для разделения по датам

```
CREATE TABLE sensor_data_2023 (
    CHECK (record_date >= '2023-01-01' AND record_date < '2024-01-01')
) INHERITS (sensor_data);
```

```
CREATE TABLE sensor_data_2024 (
```

```
CHECK (record_date >= '2024-01-01' AND record_date < '2025-01-01')
) INHERITS (sensor_data);
```

-- Таблица словаря для подмены значений

```
CREATE TABLE value_mapping (
    original_value REAL,
    mapped_value REAL
);
```

```
INSERT INTO value_mapping VALUES
(10.5, 100.0),
(20.3, 200.0),
(30.7, 300.0);
```

```
postgres=# -- Таблица с исходными значениями
postgres=# CREATE TABLE sensor_data (
postgres(#     id SERIAL PRIMARY KEY,
postgres(#     sensor_id INTEGER,
postgres(#     sensor_value REAL,
postgres(#     record_date TIMESTAMP
postgres(# );
CREATE TABLE
postgres=# -- Таблица для агрегированных данных
postgres=# CREATE TABLE aggregated_data (
postgres(#     sensor_id INTEGER PRIMARY KEY,
postgres(#     max_value REAL,
postgres(#     min_value REAL,
postgres(#     avg_value REAL,
postgres(#     last_update TIMESTAMP
postgres(# );
CREATE TABLE
postgres=# -- Таблица для разделения по датам
postgres=# CREATE TABLE sensor_data_2023 (
postgres(#     CHECK (record_date >= '2023-01-01' AND record_date < '2024-01-01')
postgres(# ) INHERITS (sensor_data);
CREATE TABLE
postgres=# -- CREATE TABLE sensor_data_2024 (
postgres(#     CHECK (record_date >= '2024-01-01' AND record_date < '2025-01-01')
postgres(# ) INHERITS (sensor_data);
CREATE TABLE
postgres=# -- Таблица словаря для подмены значений
postgres=# CREATE TABLE value_mapping (
postgres(#     original_value REAL,
postgres(#     mapped_value REAL
postgres(# );
CREATE TABLE
postgres=# INSERT INTO value_mapping VALUES
postgres=# (10.5, 100.0),
postgres=# (20.3, 200.0),
postgres=# (30.7, 300.0);
INSERT 0 3
```

## 2) Процедура для агрегации данных

```
CREATE OR REPLACE FUNCTION update_aggregated_data()
RETURNS VOID AS $$

BEGIN

    -- Удаляем старые агрегированные данные
    DELETE FROM aggregated_data;

    -- Вставляем новые агрегированные данные
    INSERT INTO aggregated_data
        SELECT
            sensor_id,
            MAX(sensor_value) AS max_value,
            MIN(sensor_value) AS min_value,
            AVG(sensor_value) AS avg_value,
            NOW() AS last_update
        FROM
            sensor_data
        GROUP BY
            sensor_id;

    RAISE NOTICE 'Aggregated data updated at %', NOW();

END;

$$ LANGUAGE plpgsql;
```

```

postgres=# CREATE OR REPLACE FUNCTION update_aggregated_data()
postgres=# RETURNS VOID AS $$ 
postgres$# BEGIN
postgres$#     -- Удаляем старые агрегированные данные
postgres$#     DELETE FROM aggregated_data;
postgres$# 
postgres$#     -- Вставляем новые агрегированные данные
postgres$#     INSERT INTO aggregated_data
postgres$#     SELECT
postgres$#         sensor_id,
postgres$#         MAX(sensor_value) AS max_value,
postgres$#         MIN(sensor_value) AS min_value,
postgres$#         AVG(sensor_value) AS avg_value,
postgres$#         NOW() AS last_update
postgres$#     FROM
postgres$#         sensor_data
postgres$#     GROUP BY
postgres$#         sensor_id;
postgres$#     RAISE NOTICE 'Aggregated data updated at %', NOW();
postgres$# END;
postgres$# $$ LANGUAGE plpgsql;
CREATE FUNCTION

```

### 3) Триггер для автоматического обновления агрегированных данных

-- Сначала создаем триггерную функцию

```
CREATE OR REPLACE FUNCTION trigger_update_aggregated()
```

```
RETURNS TRIGGER AS $$
```

```
BEGIN
```

```
    PERFORM update_aggregated_data();
```

```
    RETURN NULL;
```

```
END;
```

```
$$ LANGUAGE plpgsql;
```

-- Затем создаем сам триггер

```
CREATE TRIGGER trg_update_aggregated
```

```
AFTER INSERT OR UPDATE OR DELETE ON sensor_data
```

```
FOR EACH STATEMENT
```

```
EXECUTE PROCEDURE trigger_update_aggregated();
```

```

postgres=# -- Сначала создаем триггерную функцию
postgres=# CREATE OR REPLACE FUNCTION trigger_update_aggregated()
postgres-# RETURNS TRIGGER AS $$
postgres$# BEGIN
postgres$#     PERFORM update_aggregated_data();
postgres$#     RETURN NULL;
postgres$# END;
postgres$# $$ LANGUAGE plpgsql;
CREATE FUNCTION
postgres=#
postgres=# -- Затем создаем сам триггер
postgres=# CREATE TRIGGER trg_update_aggregated
postgres-# AFTER INSERT OR UPDATE OR DELETE ON sensor_data
postgres-# FOR EACH STATEMENT
postgres-# EXECUTE PROCEDURE trigger_update_aggregated();
CREATE TRIGGER

```

#### 4) Триггер для вставки в соответствующую таблицу по дате

```

CREATE OR REPLACE FUNCTION trigger_insert_by_date()
RETURNS TRIGGER AS $$

BEGIN

    IF NEW.record_date >= '2023-01-01' AND NEW.record_date < '2024-01-01' THEN
        INSERT INTO sensor_data_2023 VALUES (NEW.*);
    ELSIF NEW.record_date >= '2024-01-01' AND NEW.record_date < '2025-01-01' THEN
        INSERT INTO sensor_data_2024 VALUES (NEW.*);
    ELSE
        RAISE EXCEPTION 'Date out of range. No partition for year %',
                        EXTRACT(YEAR FROM NEW.record_date);
    END IF;

    RETURN NULL; -- Отменяем вставку в основную таблицу
END;
$$ LANGUAGE plpgsql;

```

```

CREATE TRIGGER trg_insert_by_date
BEFORE INSERT ON sensor_data
FOR EACH ROW
EXECUTE PROCEDURE trigger_insert_by_date();

```

```

postgres=# CREATE OR REPLACE FUNCTION trigger_insert_by_date()
postgres-# RETURNS TRIGGER AS $$
postgres$# BEGIN
postgres$#     IF NEW.record_date >= '2023-01-01' AND NEW.record_date < '2024-01-01' THEN
postgres$#         INSERT INTO sensor_data_2023 VALUES (NEW.*);
postgres$#     ELSIF NEW.record_date >= '2024-01-01' AND NEW.record_date < '2025-01-01' THEN
postgres$#         INSERT INTO sensor_data_2024 VALUES (NEW.*);
postgres$#     ELSE
postgres$#         RAISE EXCEPTION 'Date out of range. No partition for year %',
postgres$#                             EXTRACT(YEAR FROM NEW.record_date);
postgres$#     END IF;
postgres$#
postgres$#     RETURN NULL; -- Отменяем вставку в основную таблицу
postgres$# END;
postgres$# $$ LANGUAGE plpgsql;
CREATE FUNCTION
postgres=#
postgres=# CREATE TRIGGER trg_insert_by_date
postgres-# BEFORE INSERT ON sensor_data
postgres-# FOR EACH ROW
postgres-# EXECUTE PROCEDURE trigger_insert_by_date();
CREATE TRIGGER

```

## 5) Триггер для подмены значений по словарю

CREATE OR REPLACE FUNCTION trigger\_map\_values()

RETURNS TRIGGER AS \$\$

DECLARE

    mapped\_val REAL;

BEGIN

    SELECT mapped\_value INTO mapped\_val  
 FROM value\_mapping  
 WHERE original\_value = NEW.sensor\_value;

IF FOUND THEN

    NEW.sensor\_value := mapped\_val;  
 RAISE NOTICE 'Mapped value from % to %',  
 OLD.sensor\_value, NEW.sensor\_value;

END IF;

RETURN NEW;

END;

\$\$ LANGUAGE plpgsql;

CREATE TRIGGER trg\_map\_values

```
BEFORE INSERT OR UPDATE ON sensor_data
```

```
FOR EACH ROW
```

```
EXECUTE PROCEDURE trigger_map_values();
```

```
postgres=# CREATE OR REPLACE FUNCTION trigger_map_values()
postgres=# RETURNS TRIGGER AS $$
postgres$# DECLARE
postgres$#     mapped_val REAL;
postgres$# BEGIN
postgres$#     SELECT mapped_value INTO mapped_val
postgres$#     FROM value_mapping
postgres$#     WHERE original_value = NEW.sensor_value;
postgres$#
postgres$#     IF FOUND THEN
postgres$#         NEW.sensor_value := mapped_val;
postgres$#         RAISE NOTICE 'Mapped value from % to %',
postgres$#                         OLD.sensor_value, NEW.sensor_value;
postgres$#     END IF;
postgres$#
postgres$#     RETURN NEW;
postgres$# END;
postgres$# $$ LANGUAGE plpgsql;
CREATE FUNCTION
postgres=#
postgres=# CREATE TRIGGER trg_map_values
postgres=# BEFORE INSERT OR UPDATE ON sensor_data
postgres=# FOR EACH ROW
postgres=# EXECUTE PROCEDURE trigger_map_values();
CREATE TRIGGER
```

## 6) Триггер с использованием специальных переменных

```
CREATE OR REPLACE FUNCTION trigger_log_operations()
```

```
RETURNS TRIGGER AS $$
```

```
BEGIN
```

```
    IF TG_OP = 'INSERT' THEN
```

```
        RAISE NOTICE 'Inserting new record with ID %, value %, date %',
```

```
            NEW.id, NEW.sensor_value, NEW.record_date;
```

```
    ELSIF TG_OP = 'UPDATE' THEN
```

```
        RAISE NOTICE 'Updating record ID %. Old value: %, new value: %',
```

```
            NEW.id, OLD.sensor_value, NEW.sensor_value;
```

```
    ELSIF TG_OP = 'DELETE' THEN
```

```
        RAISE NOTICE 'Deleting record ID % with value %',
```

```
            OLD.id, OLD.sensor_value;
```

```
END IF;
```

```
IF TG_WHEN = 'BEFORE' THEN  
    RAISE NOTICE 'This is a BEFORE trigger';  
ELSE  
    RAISE NOTICE 'This is an AFTER trigger';  
END IF;
```

```
RAISE NOTICE 'Trigger name: %, Table name: %',  
TG_NAME, TG_TABLE_NAME;
```

```
RETURN NEW;
```

```
END;
```

```
$$ LANGUAGE plpgsql;
```

```
CREATE TRIGGER trg_log_insert  
BEFORE INSERT ON sensor_data  
FOR EACH ROW  
EXECUTE PROCEDURE trigger_log_operations();
```

```
CREATE TRIGGER trg_log_update  
BEFORE UPDATE ON sensor_data  
FOR EACH ROW  
EXECUTE PROCEDURE trigger_log_operations();
```

```
CREATE TRIGGER trg_log_delete  
BEFORE DELETE ON sensor_data  
FOR EACH ROW  
EXECUTE PROCEDURE trigger_log_operations();
```

```

postgres=# CREATE OR REPLACE FUNCTION trigger_log_operations()
postgres-# RETURNS TRIGGER AS $$ 
postgres$# BEGIN
postgres$#     IF TG_OP = 'INSERT' THEN
postgres$#         RAISE NOTICE 'Inserting new record with ID %, value %, date %',
postgres$#                         NEW.id, NEW.sensor_value, NEW.record_date;
postgres$#     ELSIF TG_OP = 'UPDATE' THEN
postgres$#         RAISE NOTICE 'Updating record ID %. Old value: %, new value: %',
postgres$#                         NEW.id, OLD.sensor_value, NEW.sensor_value;
postgres$#     ELSIF TG_OP = 'DELETE' THEN
postgres$#         RAISE NOTICE 'Deleting record ID % with value %',
postgres$#                         OLD.id, OLD.sensor_value;
postgres$#     END IF;
postgres$#
postgres$#     IF TG_WHEN = 'BEFORE' THEN
postgres$#         RAISE NOTICE 'This is a BEFORE trigger';
postgres$#     ELSE
postgres$#         RAISE NOTICE 'This is an AFTER trigger';
postgres$#     END IF;
postgres$#
postgres$#     RAISE NOTICE 'Trigger name: %, Table name: %',
postgres$#                         TG_NAME, TG_TABLE_NAME;
postgres$#
postgres$#     RETURN NEW;
postgres$# END;
postgres$# $$ LANGUAGE plpgsql;
CREATE FUNCTION
postgres=#
postgres=# CREATE TRIGGER trg_log_insert
postgres-# BEFORE INSERT ON sensor_data
postgres-# FOR EACH ROW
postgres-# EXECUTE PROCEDURE trigger_log_operations();
CREATE TRIGGER
postgres=#
postgres=# CREATE TRIGGER trg_log_update
postgres-# BEFORE UPDATE ON sensor_data
postgres-# FOR EACH ROW
postgres-# EXECUTE PROCEDURE trigger_log_operations();
CREATE TRIGGER
postgres=#
postgres=# CREATE TRIGGER trg_log_delete
postgres-# BEFORE DELETE ON sensor_data
postgres-# FOR EACH ROW
postgres-# EXECUTE PROCEDURE trigger_log_operations();
CREATE TRIGGER

```

## 7) Тестирование работы

-- Вставка тестовых данных

```

INSERT INTO sensor_data (sensor_id, sensor_value, record_date) VALUES
(1, 10.5, '2023-05-10'),
(1, 20.3, '2023-06-15'),
(2, 30.7, '2024-02-20'),
(2, 15.0, '2024-03-25');

```

-- Проверка агрегированных данных

```
SELECT * FROM aggregated_data;
```

-- Обновление данных

```
UPDATE sensor_data SET sensor_value = 25.0 WHERE sensor_value = 20.3;
```

-- Проверка обновленных агрегированных данных

```
SELECT * FROM aggregated_data;
```

-- Проверка разделения по датам

```
SELECT * FROM sensor_data_2023;
```

```
SELECT * FROM sensor_data_2024;
```

-- Проверка подмены значений

```
INSERT INTO sensor_data (sensor_id, sensor_value, record_date) VALUES
```

```
(3, 10.5, '2023-07-01'); -- Должно быть заменено на 100.0
```

```
postgres=# -- Вставка тестовых данных
postgres=# INSERT INTO sensor_data (sensor_id, sensor_value, record_date) VALUES
postgres-# (1, 10.5, '2023-05-10'),
postgres-# (1, 20.3, '2023-06-15'),
postgres-# (2, 30.7, '2024-02-20'),
postgres-# (2, 15.0, '2024-03-25');
NOTICE: Aggregated data updated at 2025-04-01 13:55:22.386302+00
INSERT 0 0
```

```

postgres=# -- Проверка агрегированных данных
postgres=# SELECT * FROM aggregated_data;
 sensor_id | max_value | min_value | avg_value |           last_update
-----+-----+-----+-----+-----+
      2 |    30.7 |      15 |    22.85 | 2025-04-01 13:55:22.386302
      1 |    20.3 |     10.5 |     15.4 | 2025-04-01 13:55:22.386302
(2 rows)

postgres=#
postgres=# -- Обновление данных
postgres=# UPDATE sensor_data SET sensor_value = 25.0 WHERE sensor_value = 20.3;
NOTICE: Aggregated data updated at 2025-04-01 13:55:22.430592+00
UPDATE 0
postgres=#
postgres=# -- Проверка обновленных агрегированных данных
postgres=# SELECT * FROM aggregated_data;
 sensor_id | max_value | min_value | avg_value |           last_update
-----+-----+-----+-----+-----+
      2 |    30.7 |      15 |    22.85 | 2025-04-01 13:55:22.430592
      1 |    20.3 |     10.5 |     15.4 | 2025-04-01 13:55:22.430592
(2 rows)

postgres=#
postgres=# -- Проверка разделения по датам
postgres=# SELECT * FROM sensor_data_2023;
 id | sensor_id | sensor_value |           record_date
----+-----+-----+-----+
    1 |        1 |      10.5 | 2023-05-10 00:00:00
    2 |        1 |      20.3 | 2023-06-15 00:00:00
(2 rows)

postgres=# SELECT * FROM sensor_data_2024;
 id | sensor_id | sensor_value |           record_date
----+-----+-----+-----+
    3 |        2 |      30.7 | 2024-02-20 00:00:00
    4 |        2 |      15 | 2024-03-25 00:00:00
(2 rows)

postgres=#
postgres=# -- Проверка подмены значений
postgres=# INSERT INTO sensor_data (sensor_id, sensor_value, record_date) VALUES
postgres-# (3, 10.5, '2023-07-01'); -- Должно быть заменено на 100.0
NOTICE: Aggregated data updated at 2025-04-01 13:56:08.891204+00
INSERT 0 0

```

Вывод: в ходе лабораторной работы были реализованы:

1. Хранимая процедура для агрегации данных
2. Триггер для автоматического обновления агрегированных данных
3. Триггер для разделения данных по таблицам в зависимости от даты
4. Триггер для подмены значений по словарю

5. Триггер с использованием специальных переменных (NEW, OLD, TG\_OP, TG\_NAME, TG\_TABLE\_NAME, TG\_WHEN)

Все триггеры и процедуры работают корректно и выполняют поставленные задачи.