Министерство науки и высшего образования Российской Федерации
Федеральное государственное автономное
образовательное учреждение высшего образования
«Национальный исследовательский университет ИТМО»

ОТЧЁТ

по лабораторной работе № 5
«Процедуры, функции, триггеры в PostgreSQL»

по дисциплине «Проектирование и реализация баз данных»

Выполнил студент: Иванов Виктор Сергеевич
Факультет прикладной информатики, группа K3240
Направление подготовки 09.03.03 Прикладная информатика
Образовательная программа «Мобильные и сетевые технологии»
Преподаватель: Говорова М. М.

Санкт-Петербург
2025

Введение

Цель работы: приобрести практические навыки создания и использования процедур, функций и триггеров в PostgreSQL :contentReference[oaicite:0]index=0:contentReference[oaicite:1]i

Практическое задание

По варианту индивидуальной БД (ЛР 2, часть IV) необходимо:

1. Создать три хранимые процедуры (CALL).
2. Разработать семь оригинальных триггеров (AFTER/BEFORE, ROW).
3. Проверить работу объектов в консоли psql, сделать скриншоты.

Схема базы данных

Наша модель «Производственный процесс» описана в ЛР 3: содержит таблицы `client`, `contract`, `detail`, . . . , `price`$_{p}eriod : contentReference[oaicite : 2]index = 2 : contentReference[oa$ 3]$index = 3$.

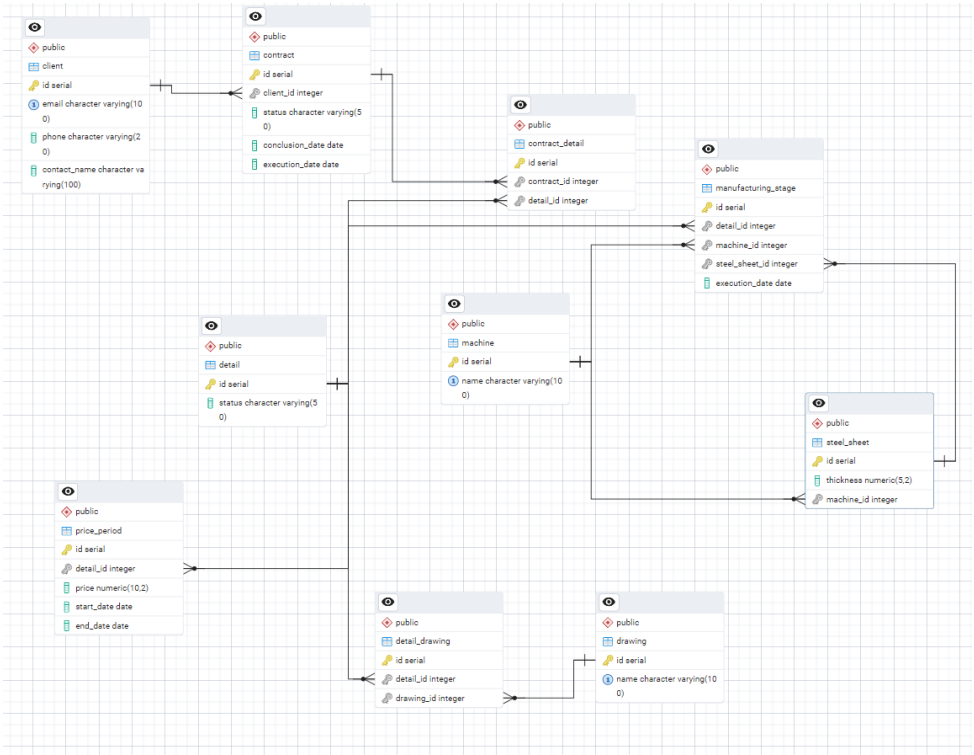

Рис. 1: ER-диаграмма базы «Производственный процесс»

Разработка хранимых процедур

1. Процедура добавления нового клиента

`Описание: процедура addClient создаёт запись в client, если почта ещё не занята.`

```sql
CREATE OR REPLACE PROCEDURE add_client(
    IN p_email      VARCHAR,
    IN p_phone      VARCHAR,
    IN p_contact    VARCHAR
)
LANGUAGE plpgsql
AS $$
BEGIN
    IF EXISTS (SELECT 1 FROM client WHERE email = p_email) THEN
        RAISE NOTICE 'Клиент с e-mail % уже существует', p_email;
    ELSE
        INSERT INTO client(email, phone, contact_name)
        VALUES(p_email, p_phone, p_contact);
    END IF;
END;
$$;
```

Листинг 1: Создание процедуры add$_client$

```
postgres=# CREATE OR REPLACE PROCEDURE add_client(
postgres(#   IN p_email   VARCHAR,
postgres(#   IN p_phone   VARCHAR,
postgres(#   IN p_contact VARCHAR
postgres(# )
postgres-# LANGUAGE plpgsql AS $$
postgres$# BEGIN
postgres$#   IF EXISTS (SELECT 1 FROM client WHERE email = p_email) THEN
postgres$#     RAISE NOTICE 'Клиент с e-mail % уже существует', p_email;
postgres$#   ELSE
postgres$#     INSERT INTO client(email, phone, contact_name)
postgres$#     VALUES (p_email, p_phone, p_contact);
postgres$#   END IF;
postgres$# END;
postgres$# $$;
CREATE PROCEDURE
postgres=# \df+ add_client

                                    Список функций
 Схема  |    Имя     | Тип данных результата |                                  Типы
 данных аргументов                                  |  Тип   | Изменчивость | Паралле
льность | Владелец | Безопасность | Права доступа |  Язык   | Внутреннее имя | Описан
ие
--------+------------+-----------------------+-----------------------------------
-------------------------------------------------+-------+--------------+--------
--------+----------+--------------+---------------+---------+----------------+-------
---
 public | add_client |                       | IN p_email character varying, IN p_pho
ne character varying, IN p_contact character varying | проц. | изменчивая   | небезоп
асная   | postgres | вызывающего  |               | plpgsql |                |
(1 строка)
```

Рис. 2: Создание процедуры `addClient`

2. Процедура открытия нового договора

Описание: openContract создаёт договор для клиента, найденного по e-mail, со статусом «в работе».

```sql
CREATE OR REPLACE PROCEDURE open_contract(
    IN p_email          VARCHAR,
    IN p_conclusion     DATE
)
```

```
LANGUAGE plpgsql
AS $$
DECLARE
    v_client_id INT;
BEGIN
    SELECT id INTO v_client_id
      FROM client WHERE email = p_email;
    IF v_client_id IS NULL THEN
        RAISE EXCEPTION 'Клиент % не найден', p_email;
    END IF;
    INSERT INTO contract(client_id, status, conclusion_date)
    VALUES(v_client_id, 'в работе', p_conclusion);
END;
$$;
```

Листинг 2: Создание процедуры open_contract

```
postgres=# CREATE OR REPLACE PROCEDURE open_contract(
postgres(#   IN p_email      VARCHAR,
postgres(#   IN p_conclusion DATE
postgres(# )
postgres-# LANGUAGE plpgsql AS $$
postgres$# DECLARE
postgres$#   v_client_id INT;
postgres$# BEGIN
postgres$#   SELECT id INTO v_client_id FROM client WHERE email = p_email;
postgres$#   IF v_client_id IS NULL THEN
postgres$#     RAISE EXCEPTION 'Клиент % не найден', p_email;
postgres$#   END IF;
postgres$#   INSERT INTO contract(client_id, status, conclusion_date)
postgres$#   VALUES (v_client_id, 'в работе', p_conclusion);
postgres$# END;
postgres$# $$;
CREATE PROCEDURE
postgres=# \df+ open_contract
```

Рис. 3: Создание процедуры openContract

3. Процедура добавления этапа изготовления

Описание: addStage для заданной детали и станка добавляет запись в manufacturingSta

```
CREATE OR REPLACE PROCEDURE add_stage(
    IN p_detail_id   INT,
    IN p_machine_id  INT,
    IN p_steel_id    INT
)
LANGUAGE plpgsql
AS $$
BEGIN
    INSERT INTO manufacturing_stage(detail_id, machine_id, steel_sheet_id,
    execution_date)
    VALUES(p_detail_id, p_machine_id, p_steel_id, CURRENT_DATE);
END;
$$;
```

Листинг 3: Создание процедуры add_stage

```
postgres=# CREATE OR REPLACE PROCEDURE add_stage(
postgres(#   IN p_detail_id  INT,
postgres(#   IN p_machine_id INT,
postgres(#   IN p_steel_id   INT
postgres(# )
postgres-# LANGUAGE plpgsql AS $$
postgres$# BEGIN
postgres$#   INSERT INTO manufacturing_stage(
postgres$#     detail_id, machine_id, steel_sheet_id, execution_date
postgres$#   ) VALUES (p_detail_id, p_machine_id, p_steel_id, CURRENT_DATE);
postgres$# END;
postgres$# $$;
CREATE PROCEDURE
```

Рис. 4: Создание процедуры `addStage`

Тестирование процедур

Тест процедуры `add_client`

```
CALL add_client(
  'xyz@example.com',
  '+70000000001',
  'Тест Т.Т.'
);
```

Листинг 4: Вызов процедуры add$_c$*lient*

```
SELECT *
FROM client
WHERE email = 'xyz@example.com';
```

Листинг 5: Проверка вставки в таблицу client

```
postgres=# CALL add_client('xyz@example.com','+70000000001','Тест Т.Т.');
CALL
postgres=# SELECT * FROM client WHERE email='xyz@example.com'\g
 id |      email      |    phone     | contact_name
----+-----------------+--------------+--------------
 19 | xyz@example.com | +70000000001 | Тест Т.Т.
(1 строка)
```

Рис. 5: Результат работы процедуры `add_client`

Тест процедуры `open_contract`

```
CALL open_contract(
  'xyz@example.com',
  CURRENT_DATE
);
```

Листинг 6: Вызов процедуры open$_c$*ontract*

```
SELECT *
FROM contract
WHERE client_id = (
  SELECT id FROM client
```

```
  WHERE email = 'xyz@example.com'
);
```

Листинг 7: Проверка вставки в таблицу contract

```
postgres=# CALL open_contract('xyz@example.com', CURRENT_DATE);
CALL
postgres=# SELECT * FROM contract
postgres-#  WHERE client_id=(SELECT id FROM client WHERE email='xyz@example.com')\g
 id  | client_id |  status  | conclusion_date | execution_date
-----+-----------+----------+-----------------+----------------
 120 |        19 | в работе | 2025-05-09      |
(1 строка)
```

Рис. 6: Результат работы процедуры `open_contract`

Тест процедуры `add_stage`

```
CALL add_stage(
  1,  -- p_detail_id
  1,  -- p_machine_id
  1   -- p_steel_id
);
```

Листинг 8: Вызов процедуры add$_stage$

```
SELECT *
FROM manufacturing_stage
WHERE execution_date = CURRENT_DATE;
```

Листинг 9: Проверка вставки в таблицу manufacturing$_stage$

```
postgres=# CALL add_stage(
postgres(#   1,  -- p_detail_id
postgres(#   1,  -- p_machine_id
postgres(#   1   -- p_steel_id
postgres(# );
CALL
postgres=# SELECT *
postgres-# FROM manufacturing_stage
postgres-# WHERE execution_date = CURRENT_DATE;
 id  | detail_id | machine_id | steel_sheet_id | execution_date
-----+-----------+------------+----------------+----------------
 318 |         1 |          1 |              1 | 2025-05-09
(1 строка)
```

Рис. 7: Результат работы процедуры `add_stage`

Разработка триггеров

Триггер 1. Авто-статус `detail` при создании этапа

Описание: после вставки в manufacturingStage меняем статус детали на «выполнена».

```sql
CREATE OR REPLACE FUNCTION fn_complete_detail() RETURNS trigger AS $$
BEGIN
  UPDATE detail
    SET status = 'выполнена'
    WHERE id = NEW.detail_id;
  RETURN NEW;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER detail_complete_trigger
AFTER INSERT ON manufacturing_stage
FOR EACH ROW
EXECUTE PROCEDURE fn_complete_detail();
```

Листинг 10: Триггер $\text{detail}_c\textit{omplete}_t\textit{rigger}$

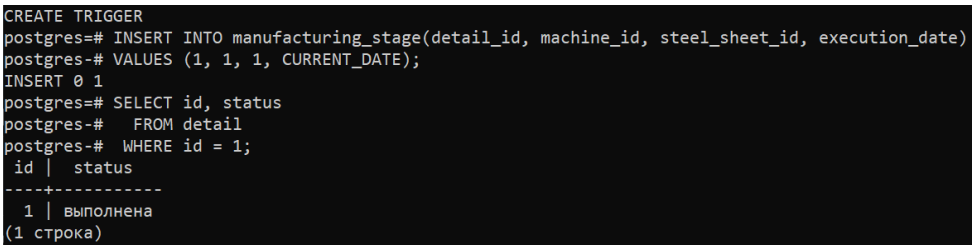

Рис. 8: Тест триггера 1: вставка этапа и проверка статуса детали

Триггер 2. Обновление статуса договора

Описание: после изменения manufacturingStage проверяем, все ли детали договора выполнены, и меняем статус договора.

```sql
CREATE OR REPLACE FUNCTION fn_finalize_contract() RETURNS trigger AS $$
BEGIN
  UPDATE contract ct
    SET status = 'завершен'
  WHERE NOT EXISTS (
    SELECT 1
      FROM contract_detail cd
      JOIN detail d ON cd.detail_id = d.id
      WHERE cd.contract_id = ct.id
        AND d.status != 'выполнена'
  );
  RETURN NEW;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER contract_finalize_trigger
AFTER INSERT OR UPDATE ON manufacturing_stage
FOR EACH ROW
EXECUTE PROCEDURE fn_finalize_contract();
```

Листинг 11: Триггер $\text{contract}_f\textit{inalize}_t\textit{rigger}$

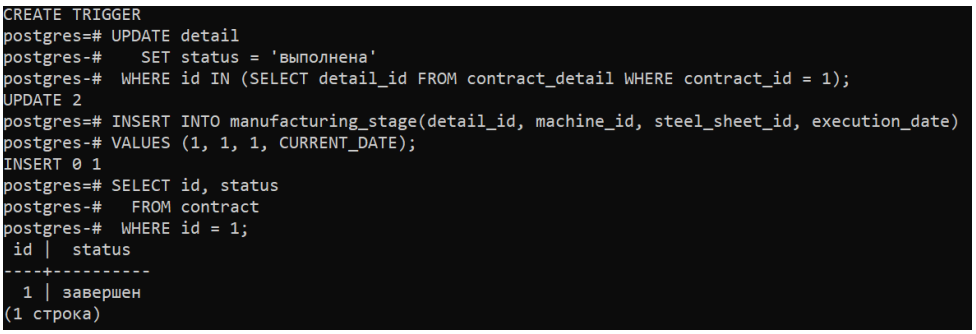

Рис. 9: Тест триггера 2: после всех этапов статус договора меняется

Триггер 3. Проверка не пересекающихся периодов цен

Описание: до вставки в pricePeriod запрещаем наложение дат.

```
CREATE OR REPLACE FUNCTION fn_check_price_period() RETURNS trigger AS $$
BEGIN
  IF EXISTS(
    SELECT 1 FROM price_period pp
    WHERE pp.detail_id = NEW.detail_id
      AND NOT (NEW.end_date < pp.start_date OR NEW.start_date > pp.end_date)
  ) THEN
    RAISE EXCEPTION 'Период цен пересекается с существующим';
  END IF;
  RETURN NEW;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER price_period_check
BEFORE INSERT ON price_period
FOR EACH ROW
EXECUTE PROCEDURE fn_check_price_period();
```

Листинг 12: Триггер $\text{price}_{p}\textit{eriod}_{c}\textit{heck}$

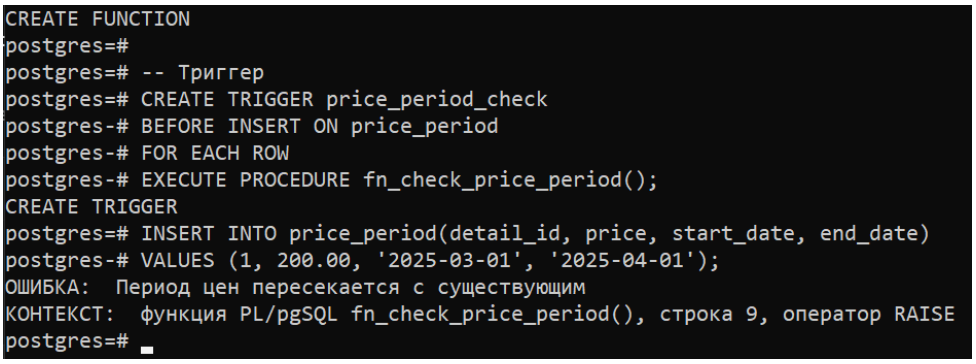

Рис. 10: Тест триггера 3: попытка вставки пересекающегося периода

Триггер 4. Валидация толщины листа

Описание: до вставки в steelSheet проверяем, что толщина > 0.1.

```
CREATE OR REPLACE FUNCTION fn_check_thickness() RETURNS trigger AS $$
```

```
BEGIN
  IF NEW.thickness <= 0.1 THEN
    RAISE EXCEPTION 'Толщина листа должна быть > 0.1';
  END IF;
  RETURN NEW;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER steel_sheet_check
BEFORE INSERT ON steel_sheet
FOR EACH ROW
EXECUTE PROCEDURE fn_check_thickness();
```

Листинг 13: Триггер $\text{steel}_s\textit{heet}_c\textit{heck}$

```
CREATE TRIGGER
postgres=# INSERT INTO steel_sheet(thickness, machine_id)
postgres-# VALUES (0.05, 1);
ОШИБКА:  Толщина листа должна быть > 0.1
КОНТЕКСТ:  функция PL/pgSQL fn_check_thickness(), строка 4, оператор RAISE
```

Рис. 11: Тест триггера 4: валидация толщины листа

Триггер 5. Установка статуса договора по умолчанию

Описание: перед вставкой в contract если статус не задан, ставим «в работе».

```
CREATE OR REPLACE FUNCTION fn_default_contract_status() RETURNS trigger AS $$
BEGIN
  IF NEW.status IS NULL THEN
    NEW.status := 'в работе';
  END IF;
  RETURN NEW;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER contract_default_status
BEFORE INSERT ON contract
FOR EACH ROW
EXECUTE PROCEDURE fn_default_contract_status();
```

Листинг 14: Триггер $\text{contract}_d\textit{efault}_s\textit{tatus}$

Триггер 6. Логирование удаления этапа

Описание: после удаления записи из manufacturingStage складываем её в auditManufact

```
CREATE TABLE audit_manufacturing_stage AS TABLE manufacturing_stage WITH NO DATA
    ;

CREATE OR REPLACE FUNCTION fn_audit_stage_delete() RETURNS trigger AS $$
BEGIN
  INSERT INTO audit_manufacturing_stage SELECT OLD.*;
  RETURN OLD;
```

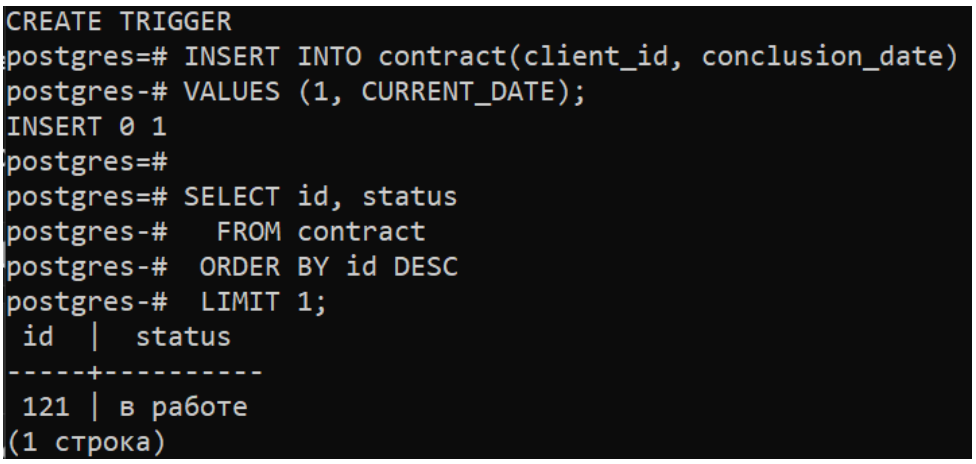

Рис. 12: Тест триггера 5: вставка договора без статуса

```
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER audit_stage_delete
AFTER DELETE ON manufacturing_stage
FOR EACH ROW
EXECUTE PROCEDURE fn_audit_stage_delete();
```

Листинг 15: Триггер audit$_{s}tage_{d}elete$

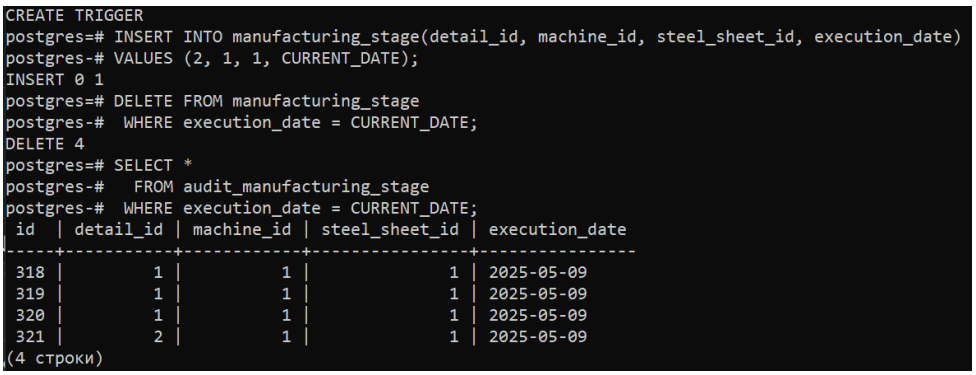

Рис. 13: Тест триггера 6: удаление этапа и запись в аудит

Триггер 7. Логирование изменений договора

Описание: после UPDATE в contract сохраняем старую и новую версию в auditContract.

```
CREATE TABLE audit_contract (
  old_id INT, old_status VARCHAR, old_exec DATE,
  new_id INT, new_status VARCHAR, new_exec DATE,
  changed_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);

CREATE OR REPLACE FUNCTION fn_audit_contract_update() RETURNS trigger AS $$
BEGIN
  INSERT INTO audit_contract
  VALUES(
```

```
    OLD.id, OLD.status, OLD.execution_date,
    NEW.id, NEW.status, NEW.execution_date
  );
  RETURN NEW;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER audit_contract_update
AFTER UPDATE ON contract
FOR EACH ROW
EXECUTE PROCEDURE fn_audit_contract_update();
```

Листинг 16: Триггер audit$_c$ontract$_u$pdate

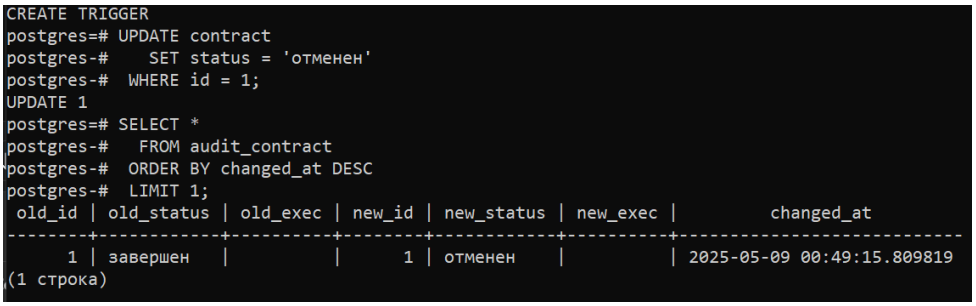

Рис. 14: Тест триггера 7: изменение договора и аудит

Выводы

В ходе выполнения работы:

- Разработаны три процедуры: добавление клиента, открытие договора, запись этапа :contentReference[oaicite:4]index=4:contentReference[oaicite:5]index=5.
- Созданы семь триггеров для автоматизации и валидации данных.
- Проверена их работоспособность в psql, результаты зафиксированы скриншотами.

Использование процедур и триггеров позволяет концентрировать логику на стороне СУБД, упрощая клиентские приложения и обеспечивая целостность данных.