Санкт-Петербургский национальный исследовательский университет
информационных технологий, механики и оптики

Факультет инфокоммуникационных технологий

Направление подготовки 11.03.02

Практическая работа №4

Выполнил:

Дощенников Никита Андреевич

Группа: К3221

Проверила:

Татьяна Евгеньевна Войтюк

Санкт-Петербург
2025

Цель работы

Освоить создание и использование обычных и материализованных представлений, пользовательских функций и хранимых процедур.

Задачи, решаемые при выполнении работы

Создать обычные и материализованные представления, протестировать их работу, изучить влияние изменений в базовых таблицах, разработать пользовательские функции, создать и вызвать хранимые процедуры, проверить корректность их выполнения.

Исходные данные

СУБД PostgreSQL, база данных `appdb`, схема “EmployeesDepartments”, таблица “EMPLOYEES” с полями сотрудника.

Выполнение работы

Задание 1. Создание представления с помощью графического интерфейса

Я открыл окно создание представления в БД `appdb` в схеме “EmployeesDepartments”. (Рис. 1)

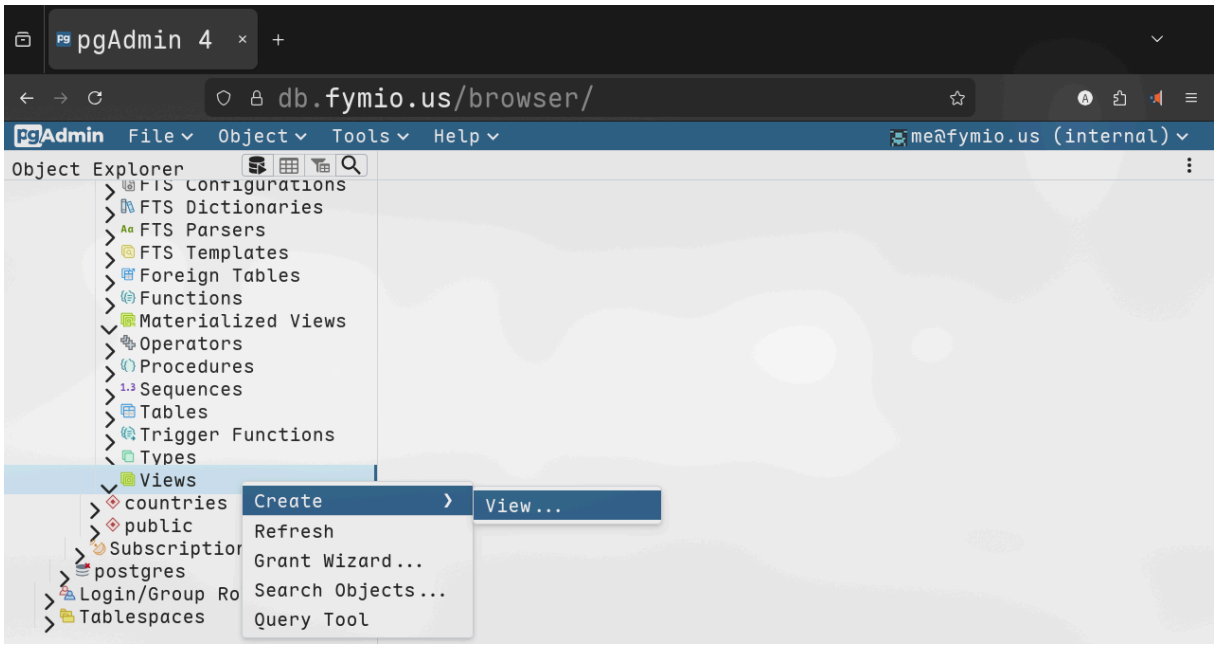

Рис. 1: Создание представления.

Ввел наименование представления `v_emp_active` на вкладке General. (Рис. 2)

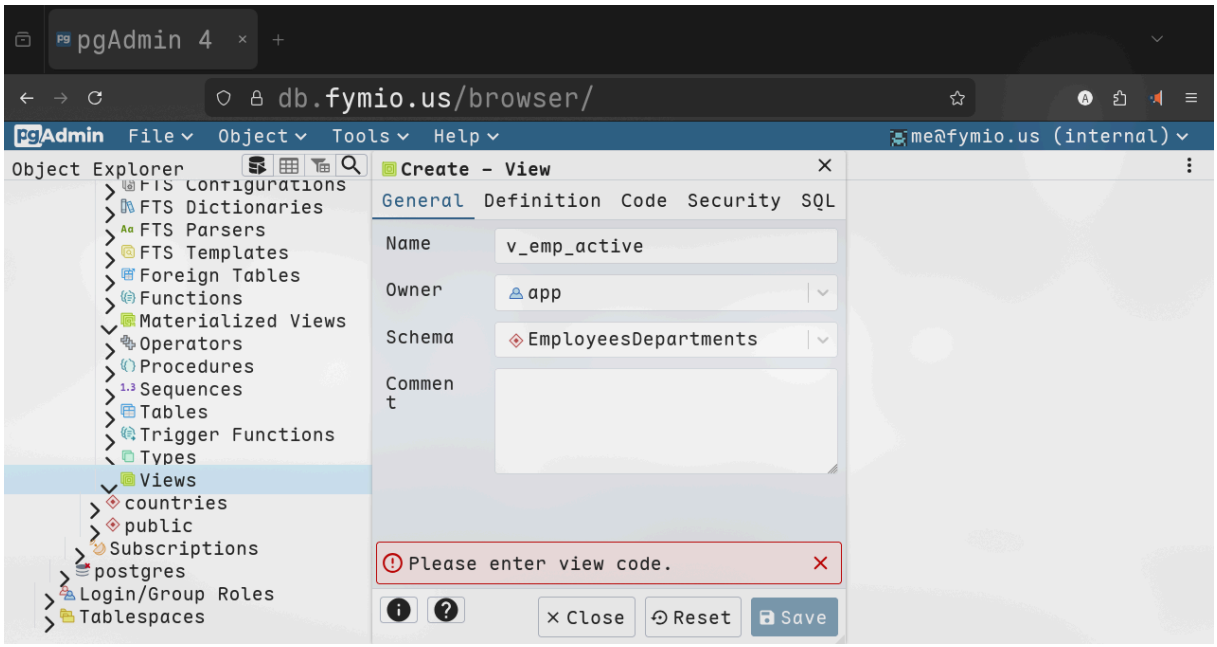

Рис. 2: Задание наименования представления.

На вкладке Code я ввел его определение. (Рис. 3)

```
SELECT "EMPLOYEE_ID",
       "FIRST_NAME",
       "LAST_NAME",
       "EMAIL",
       "JOB_ID",
       "SALARY",
       "DEPARTMENT_ID"
FROM "EmployeesDepartments"."EMPLOYEES"
WHERE "SALARY" > 5000;
```

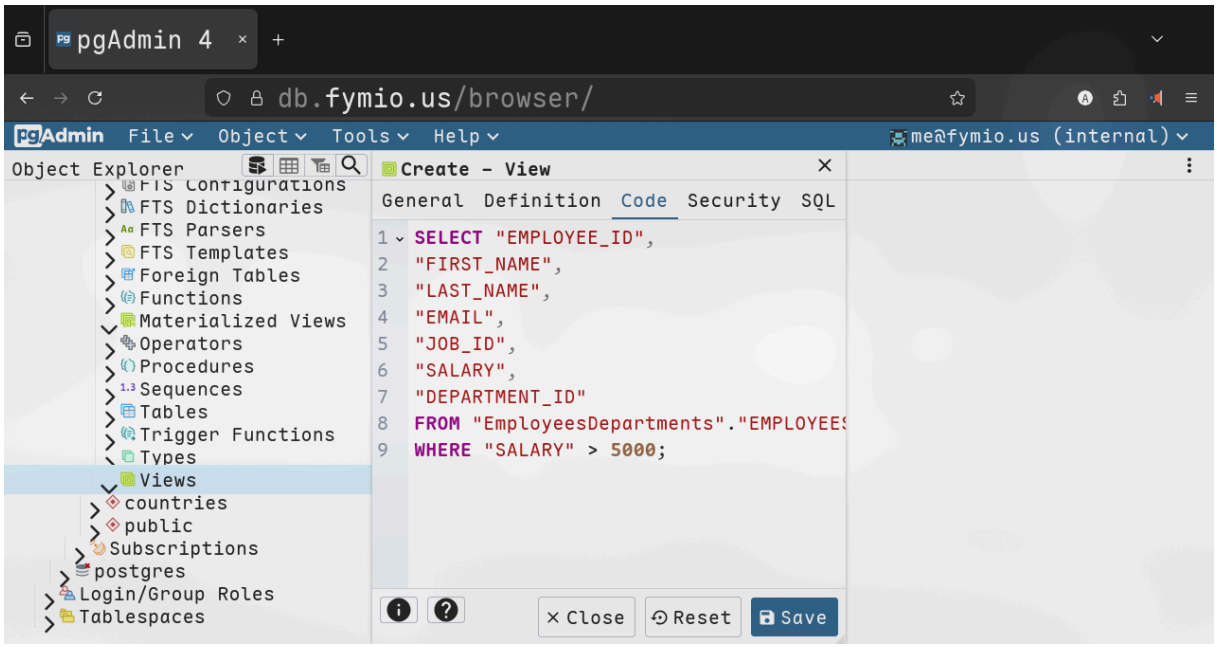

Рис. 3: Создание тела представления `v_emp_active`.

Затем я протестировал созданное представление командой ниже. (Рис. 4)

```
SELECT * FROM "EmployeesDepartments".v_emp_active
```

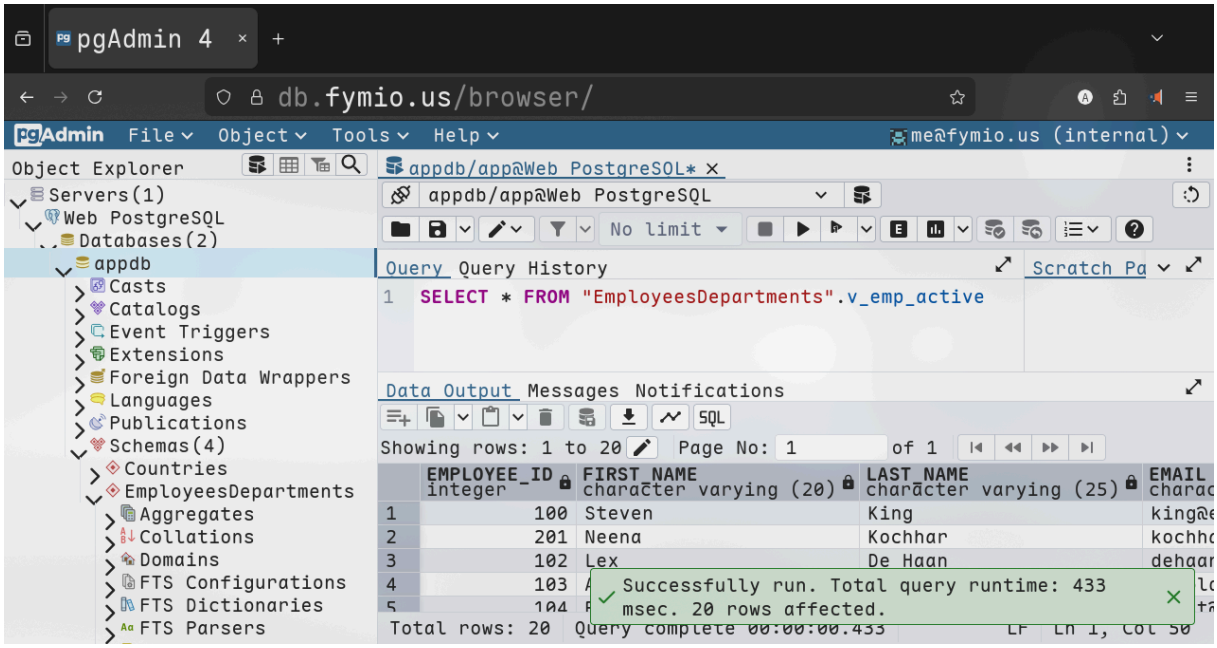

Рис. 4: Тестирование представления `v_emp_active`.

Задание 2. Создание представления на основе представления и изменение базовых таблиц представлений.

Аналогичным образом в схеме “EmployeesDepartments” я создал еще одно представление, которое имеет название `V_EMP_ACTIVE_INFO` и предназначеное для отбора сотрудников из отдела 80 с заработной платой выше 5000. Для этого я определил DDL оператор `CREATE` для создания нового представления. (Рис. 5)

```
CREATE OR REPLACE VIEW
"EmployeesDepartments"."V_EMP_ACTIVE_INFO" AS
SELECT "EMPLOYEE_ID",
       "FIRST_NAME" || ' ' || "LAST_NAME" AS FULL_NAME,
       "EMAIL",
       "JOB_ID",
       "SALARY",
       "DEPARTMENT_ID"
FROM "EmployeesDepartments".v_emp_active
WHERE "DEPARTMENT_ID" = 80;
```

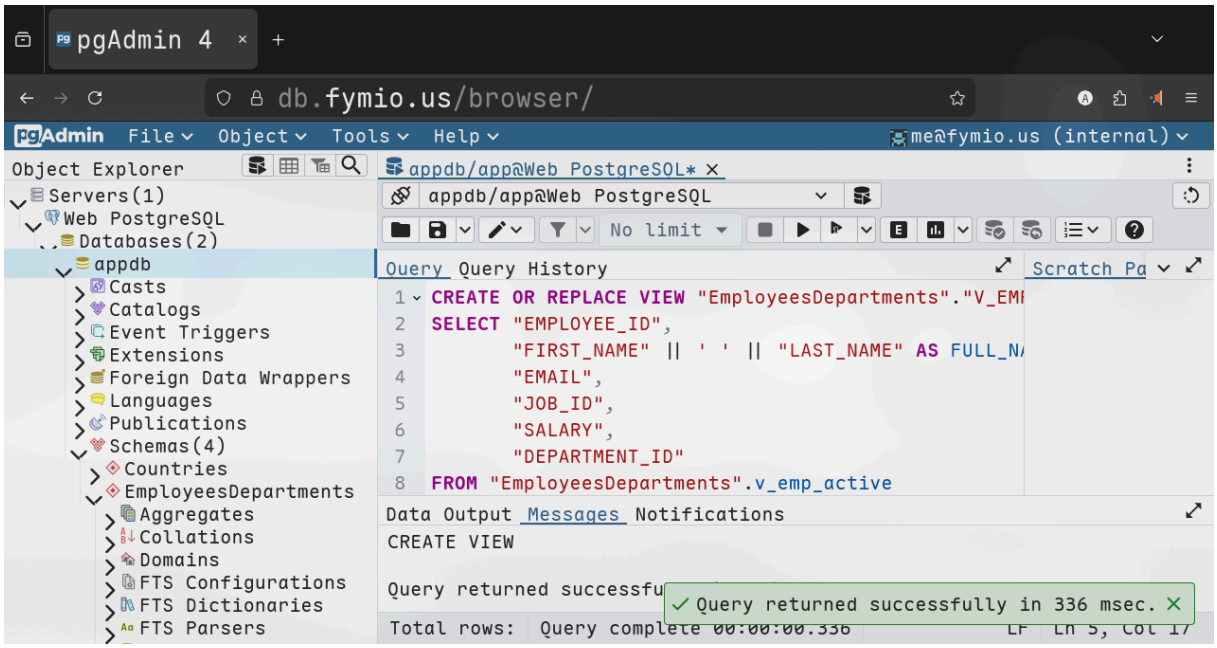

Рис. 5: Создание представления `V_EMP_ACTIVE_INFO`.

Затем, я протестировал созданное представление (Рис. 6)

```
SELECT * FROM "EmployeesDepartments"."V_EMP_ACTIVE_INFO"
```

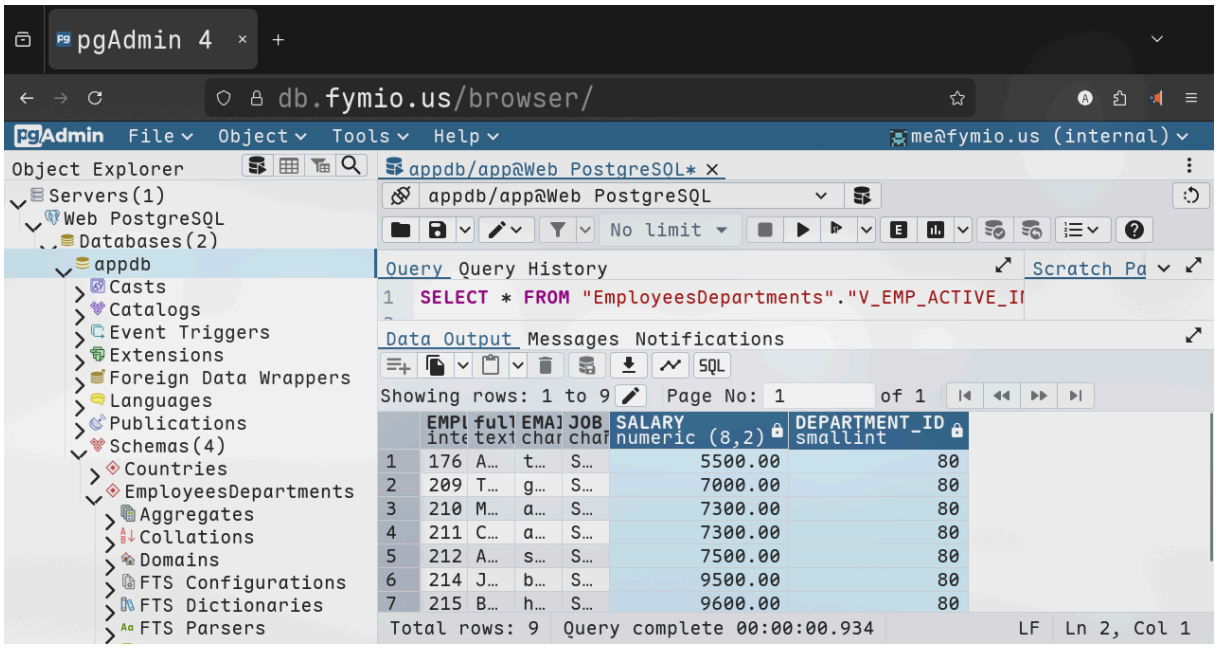

Рис. 6: Тестирование созданного представления.

После этого я провел дальнейшее тестирование представления `V_EMP_ACTIVE_INFO`. Для этого я добавил одну запись непосредственно в таблицу “EMPLOYEES”, а вторую запись добавил через представление `v_emp_active` и проверил, что представление `V_EMP_ACTIVE_INFO` отображает корректные данные. (Рис. 7)

```
INSERT INTO "EmployeesDepartments"."EMPLOYEES"
("FIRST_NAME", "LAST_NAME", "EMAIL", "JOB_ID", "SALARY",
"DEPARTMENT_ID")
VALUES('Nik', 'Nikov', 'Ni_Nik@itmo.ru', 'SA_REP', 6000, 80 );

INSERT INTO "EmployeesDepartments".v_emp_active
("FIRST_NAME", "LAST_NAME", "EMAIL", "JOB_ID", "SALARY",
"DEPARTMENT_ID")
VALUES('Sem', 'Semov', 'Se_Sem@itmo.ru', 'AD_VP', 22000, 80 );

SELECT * FROM "EmployeesDepartments"."V_EMP_ACTIVE_INFO"
```

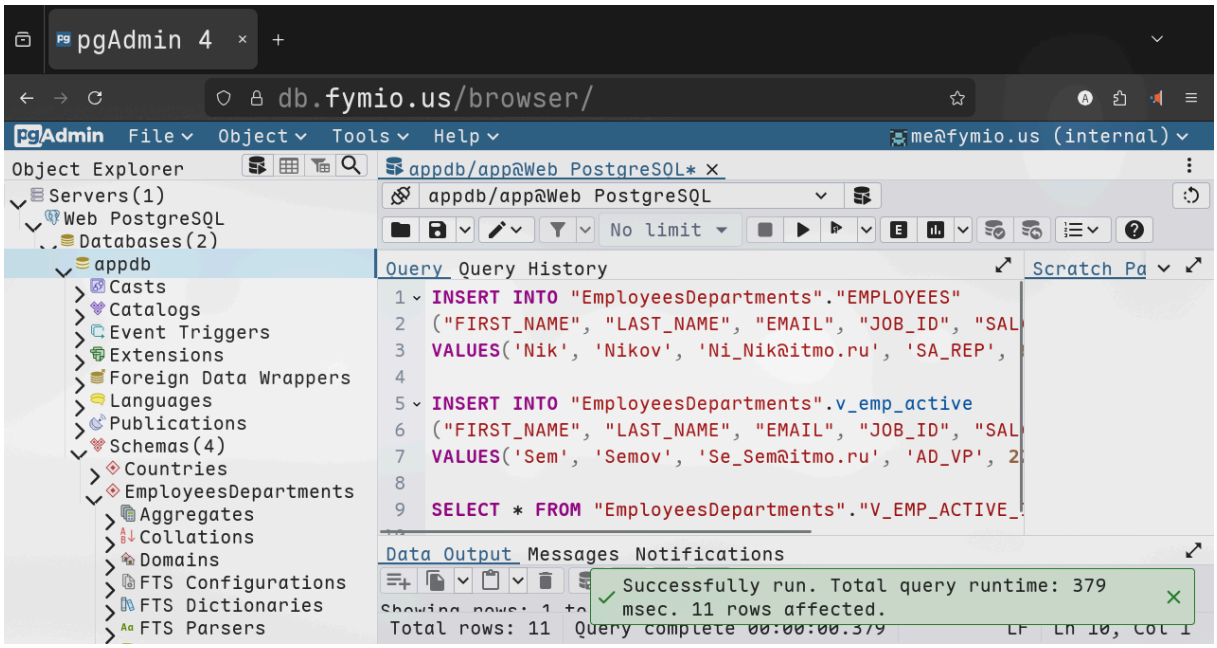

Рис. 7: Тестирование представления `V_EMP_ACTIVE_INFO`.

Затем, я попробовал внести изменения в таблицу “EMPLOYEES” с помощью кода ниже (Рис. 8)

```
ALTER TABLE "EmployeesDepartments"."EMPLOYEES"
ALTER COLUMN "FIRST_NAME" TYPE varchar(100);
```

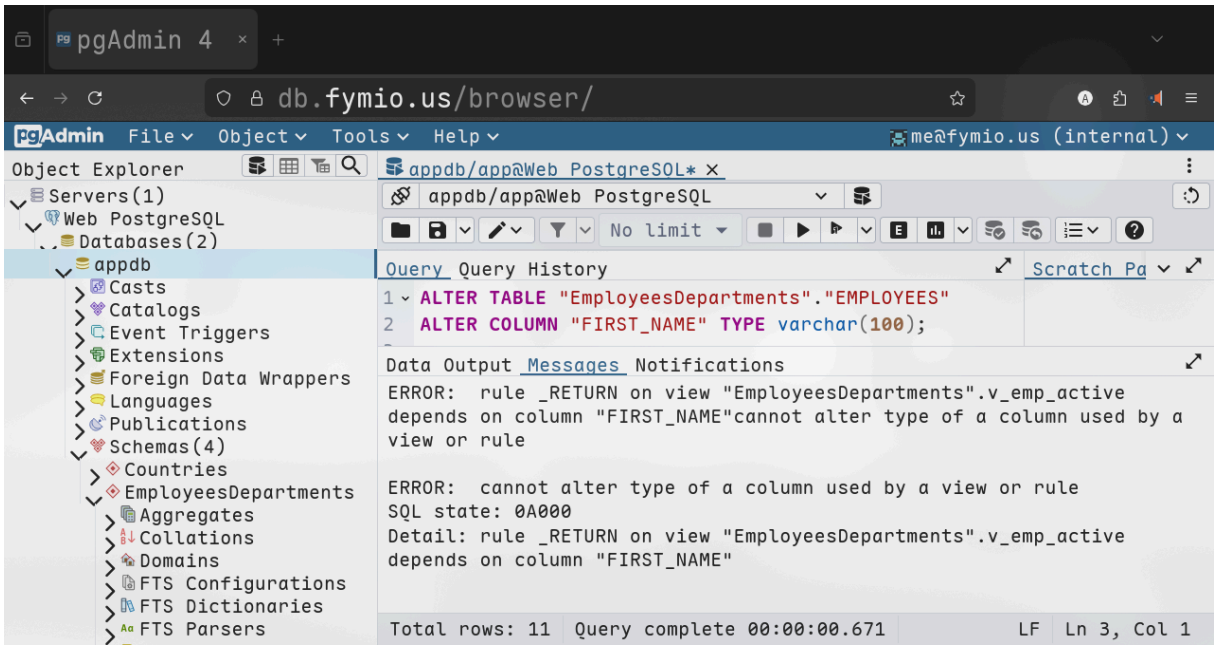

Рис. 8: Попытка изменения таблицы “EMPLOYEES”.

Я получил ошибку (Рис. 8), которая говорит о том, что мы не можем менять тип столбца, используемого представлением.

Задание 3. Создание материализованного представления

Я пересоздал существующее представление с использованием директивы `MATERIALIZED`. (Рис. 9)

```
DROP VIEW IF EXISTS "EmployeesDepartments".v_emp_active
CASCADE;

CREATE MATERIALIZED VIEW "EmployeesDepartments".v_emp_active
AS
SELECT "EMPLOYEE_ID",
       "FIRST_NAME",
       "LAST_NAME",
       "EMAIL",
       "JOB_ID",
       "SALARY",
       "DEPARTMENT_ID"
FROM "EmployeesDepartments"."EMPLOYEES"
WHERE "SALARY" > 5000;
```

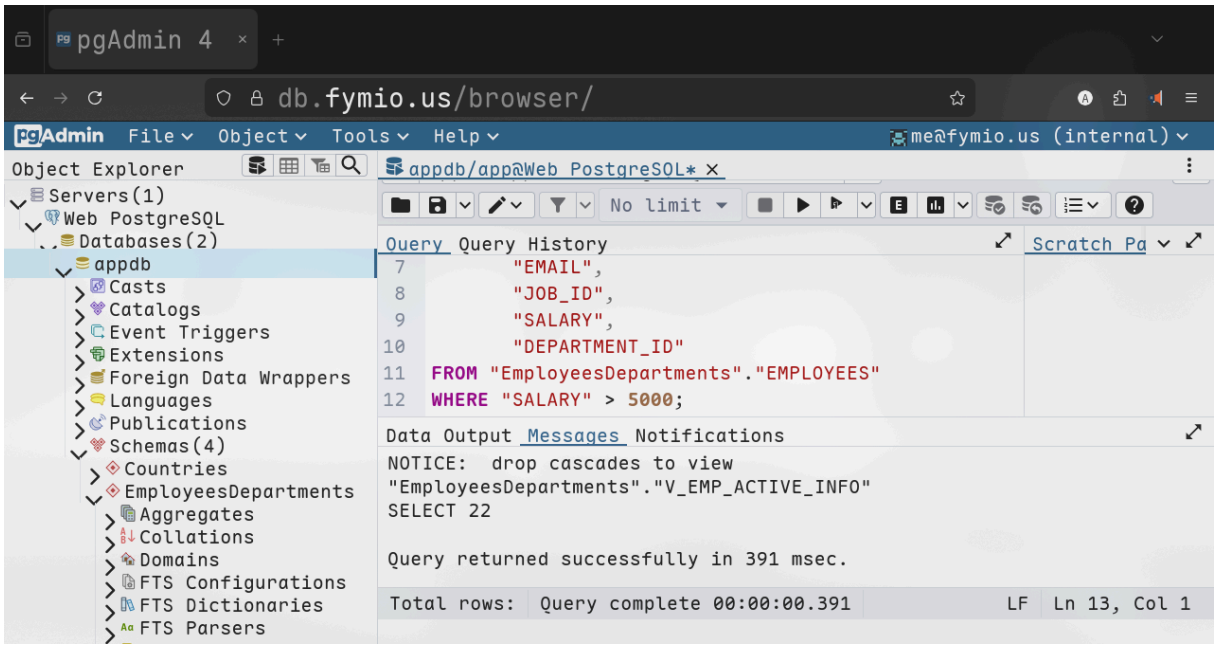

Рис. 9: Создание материализованного представления.

Затем я проверил новое представление (Рис. 10):

```
SELECT * FROM "EmployeesDepartments".v_emp_active WHERE
LAST_NAME = "Ivanov";
```

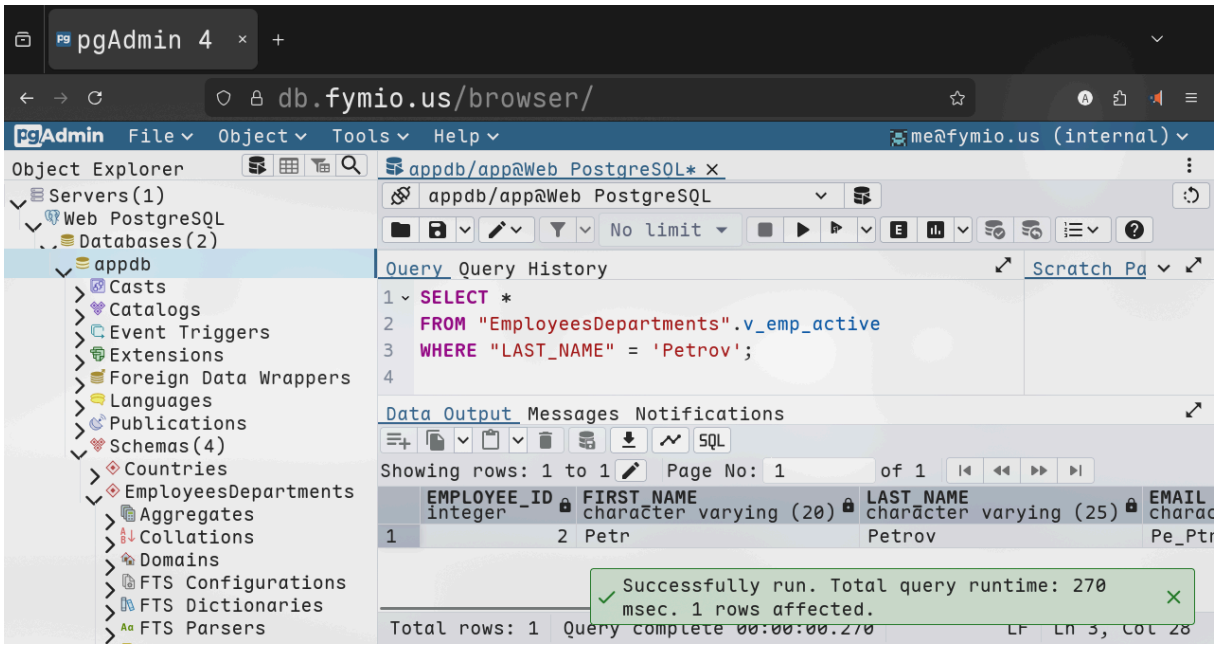

Рис. 10: Проверка представления `v_emp_active`.

Затем, я обновил имя сотрудника с фамилией “Petrov” в таблице “Employees”, установив ему имя “Vlad”. (Рис. 11)

```
UPDATE "EmployeesDepartments"."EMPLOYEES"
SET "FIRST_NAME" = 'Vlad'
WHERE "LAST_NAME" = 'Petrov';
```

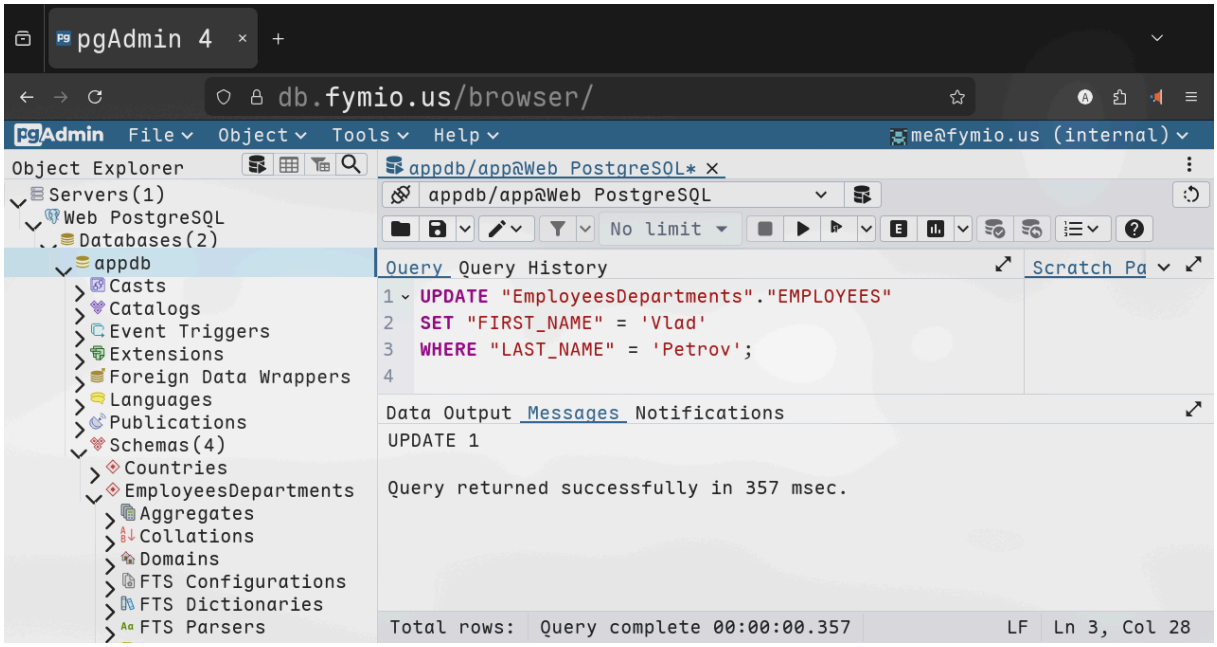

Рис. 11: Обновление имени сотрудника.

Проверим, что данные обновились в таблице “EMPLOYEES”. (Рис. 12)

```
SELECT * FROM "EmployeesDepartments"."EMPLOYEES" WHERE
"LAST_NAME" = 'Petrov';
```

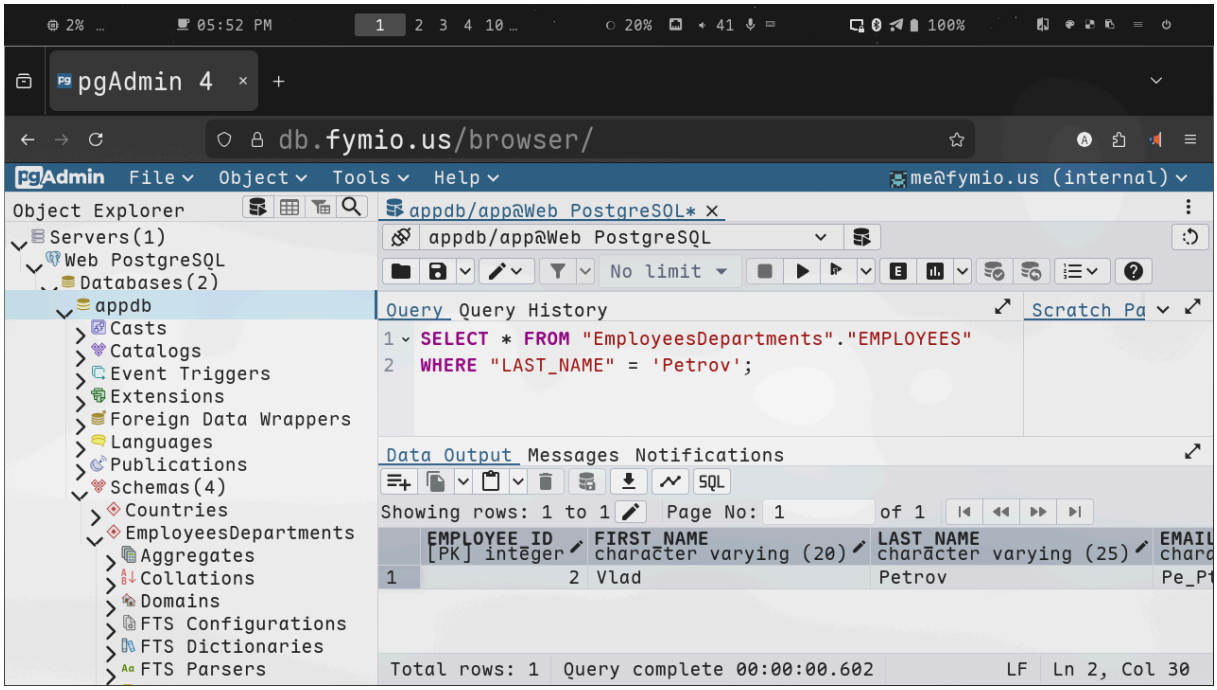

Рис. 12: Проверка обновления данных в таблице.

Чтобы получить актуальную информацию из материализованного представления, я обновил его с помощью команды `REFRESH MATERIALIZED VIEW`. (Рис. 13)

```
REFRESH MATERIALIZED VIEW "EmployeesDepartments".v_emp_active;
SELECT * FROM "EmployeesDepartments".v_emp_active
WHERE "LAST_NAME" = 'Petrov';
```

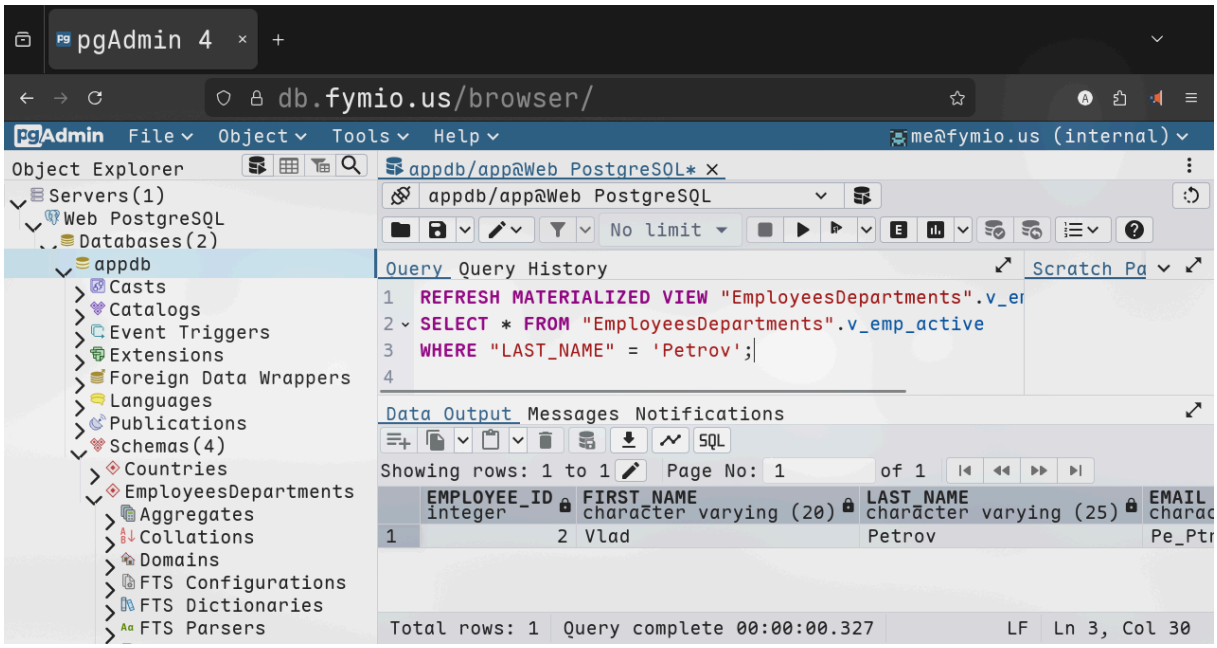

Рис. 13: Обновление представления.

Обычное представление не хранит данные, а каждый раз вычисляет результат заново, поэтому оно всегда показывает актуальную информацию. Материализованное представление хранит готовые данные, работает быстрее, но может содержать устаревшие значения и требует обновления. Обычное представление используют, когда важна актуальность данных, а материализованное — когда запрос тяжёлый, данные меняются редко и нужна высокая скорость чтения. (Рис. 14)

Задание 4. Создание скалярной пользовательской функции

В окне Query Tool я создал функцию, которая по идентификатору сотрудника, возвращает его полное имя, т.е. имя, соединенное с фамилией, через пробел. Если сотрудник не найден, то функция должна вернуть значение `NULL`.

```
CREATE OR REPLACE FUNCTION
"EmployeesDepartments".get_employee_full_name(p_emp_id INT)
RETURNS TEXT AS
$$
DECLARE
    v_full_name TEXT;
BEGIN
    SELECT "FIRST_NAME" || ' ' || "LAST_NAME"
    INTO   v_full_name
```

```
    FROM   "EmployeesDepartments"."EMPLOYEES"
    WHERE  "EMPLOYEE_ID" = p_emp_id;

    RETURN v_full_name;
END;
$$ LANGUAGE plpsql;
```

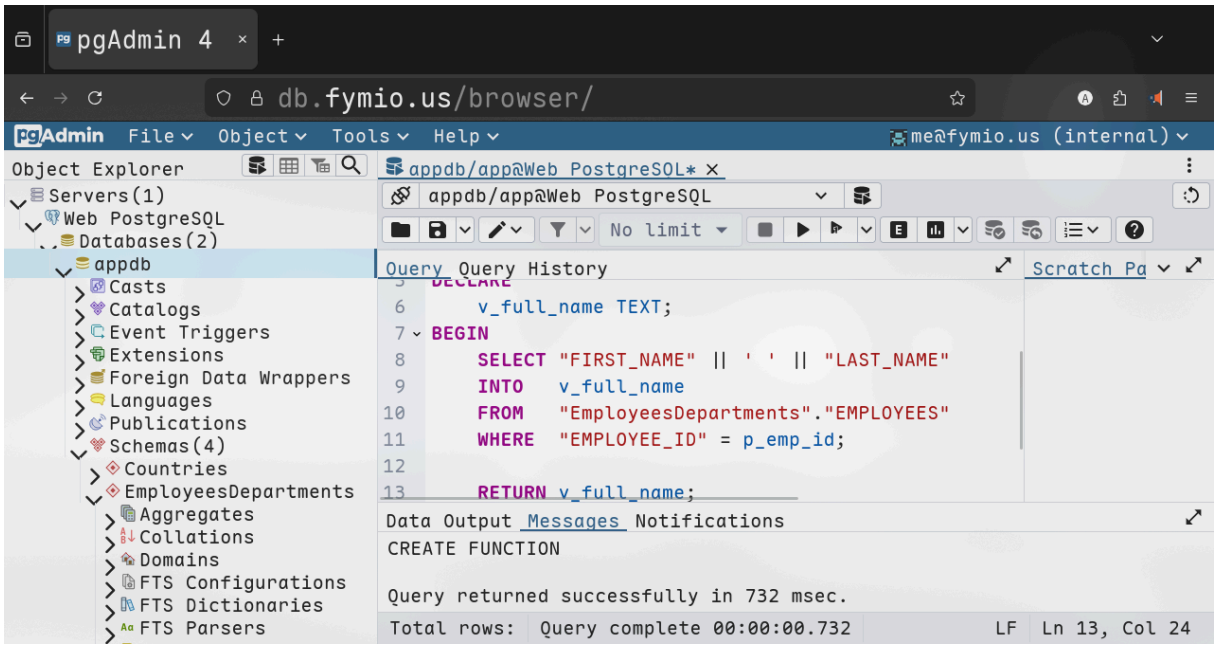

Рис. 14: Создание скалярной функции.

Затем я протестировал работоспособность скаларной функции, передав в нее известный мне идентификатор сотрудника. (Рис. 15)

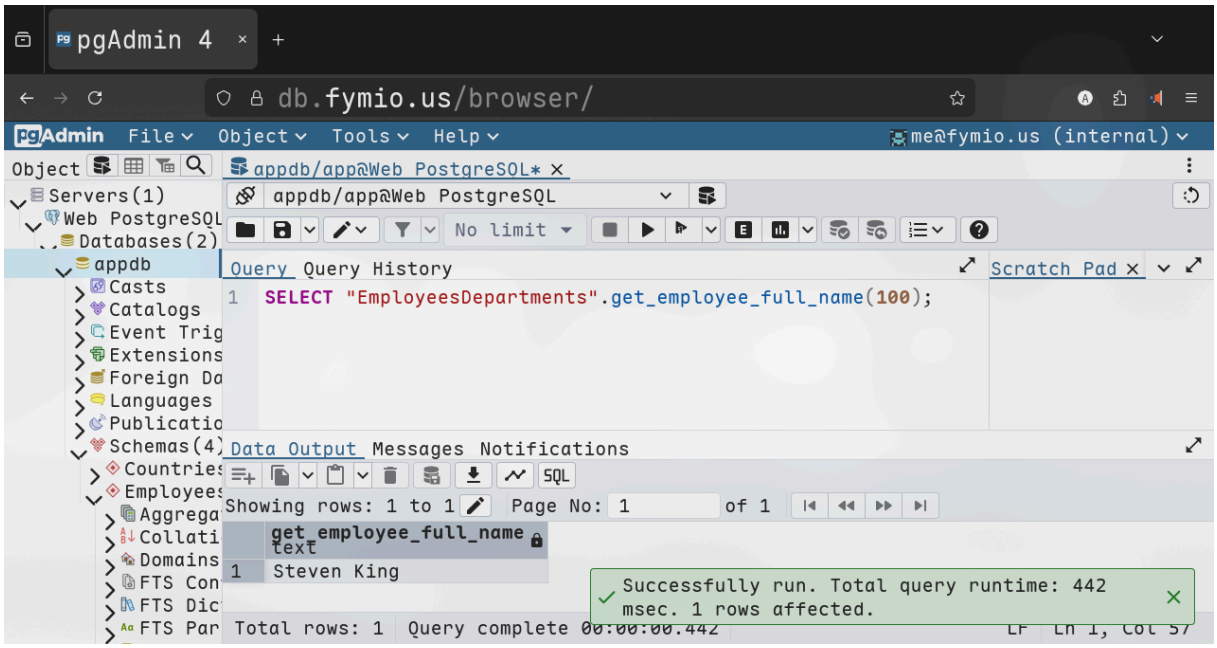

Рис. 15: Тестирование скалярной функции.

Задание 5. Создание собственной скалярной функции

Я создал собственную функцию (Рис. 16)

```
CREATE OR REPLACE FUNCTION
"EmployeesDepartments".get_salary_by_mode(p_emp_id INT, p_mode
TEXT)
RETURNS NUMERIC AS
$$
DECLARE
    v_salary NUMERIC;
BEGIN
    SELECT "SALARY" INTO v_salary
    FROM "EmployeesDepartments"."EMPLOYEES"
    WHERE "EMPLOYEE_ID" = p_emp_id;

    IF v_salary IS NULL THEN
        RETURN NULL;
    END IF;

    IF upper(p_mode) = 'ROUND' THEN
        RETURN round(v_salary);
    ELSIF upper(p_mode) = 'EXACT' THEN
        RETURN v_salary;
    ELSE
        RETURN v_salary;
```

```
    END IF;
END;
$$ LANGUAGE plpgsql;
```

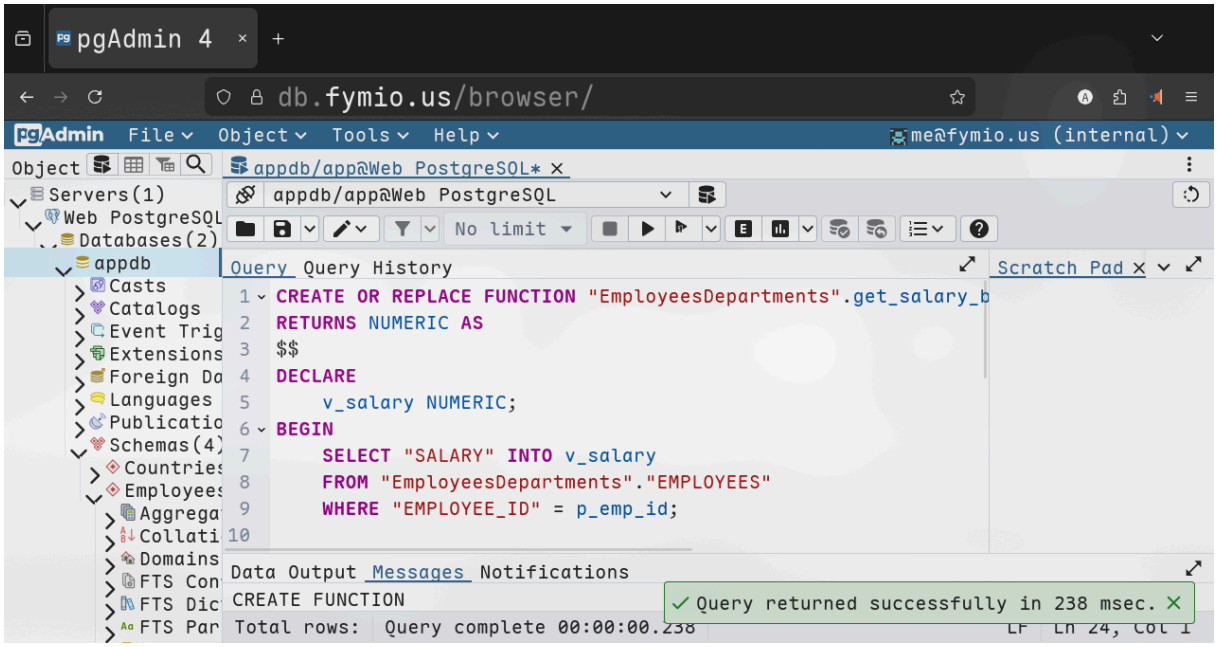

Рис. 16: Создание функции.

Точный режим. Находим сотрудника по имени и запросим точную зарплату. (Рис. 17)

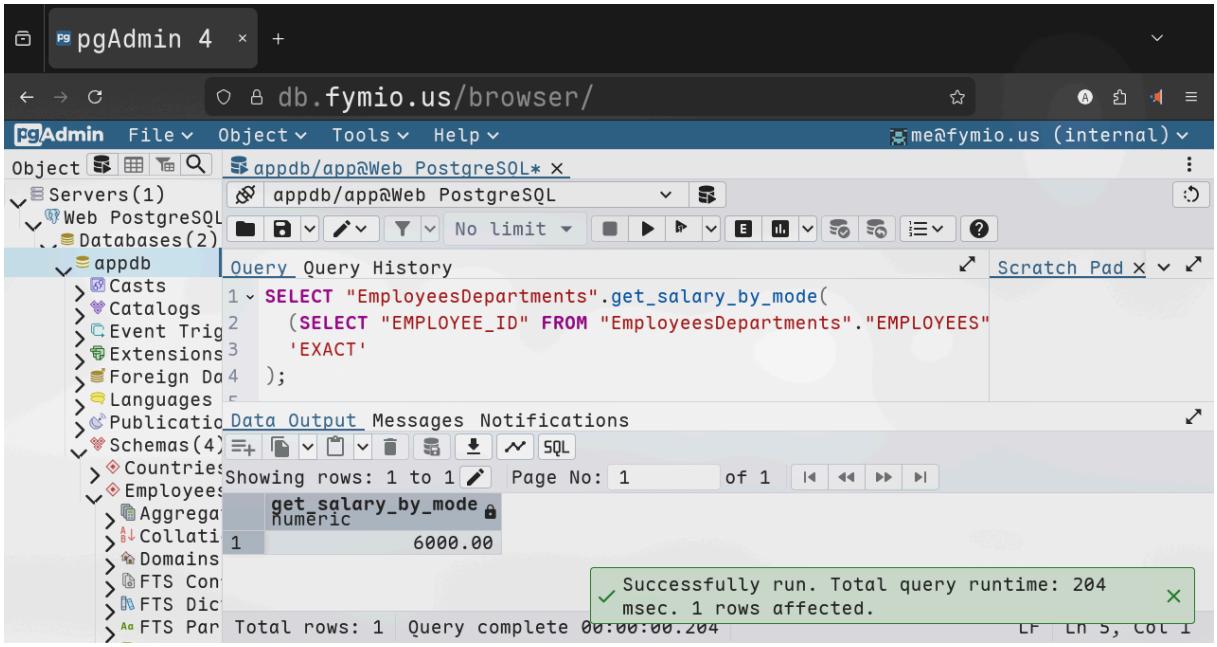

Рис. 17: Демонстрация точного режима.

Округленный режим. Округляет зарплату до целого числа. (Рис. 18)

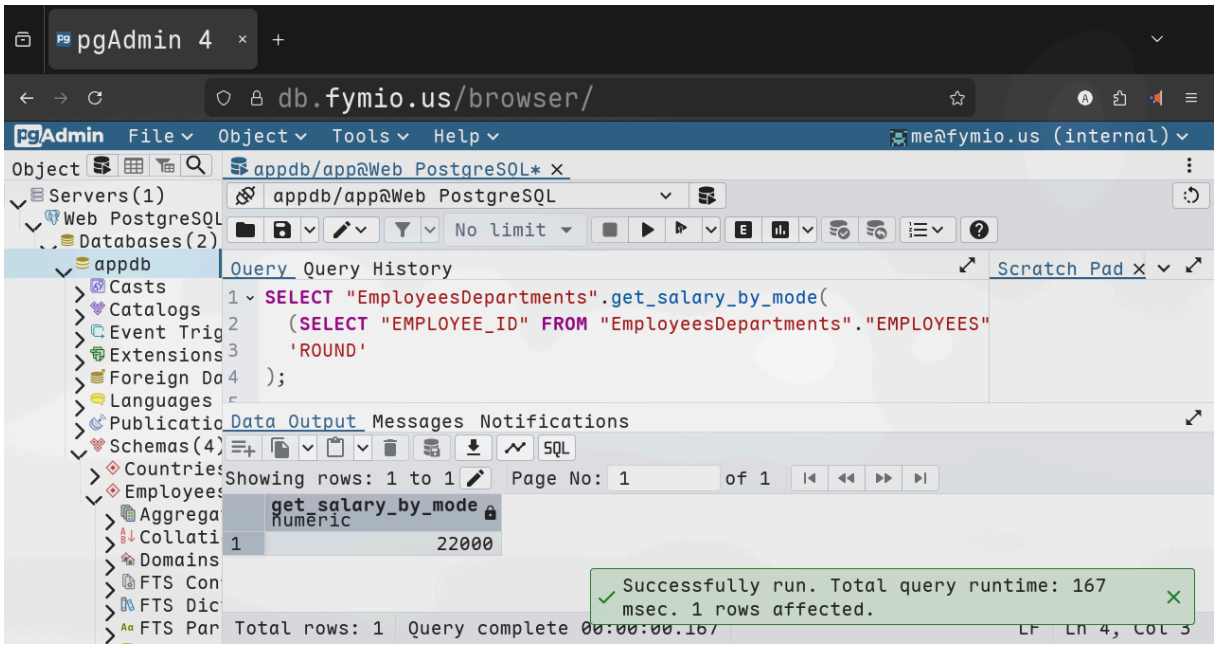

Рис. 18: Демонстрация округленного режима.

Задание 6. Создание хранимой процедуры

Я создал хранимую процедуру, предназначенную для увеличения заработной платы выбранного сотрудника на указанный процент. (Рис. 19)

```
CREATE OR REPLACE PROCEDURE
"EmployeesDepartments".raise_salary(
    p_employee_id INTEGER,
    p_percent NUMERIC
)
LANGUAGE plpgsql
AS $$
BEGIN
    -- Увеличиваем зарплату сотрудника
    UPDATE "EmployeesDepartments"."EMPLOYEES"
    SET "SALARY" = "SALARY" * (1 + p_percent / 100)
    WHERE "EMPLOYEE_ID" = p_employee_id;
END;
$$;
```

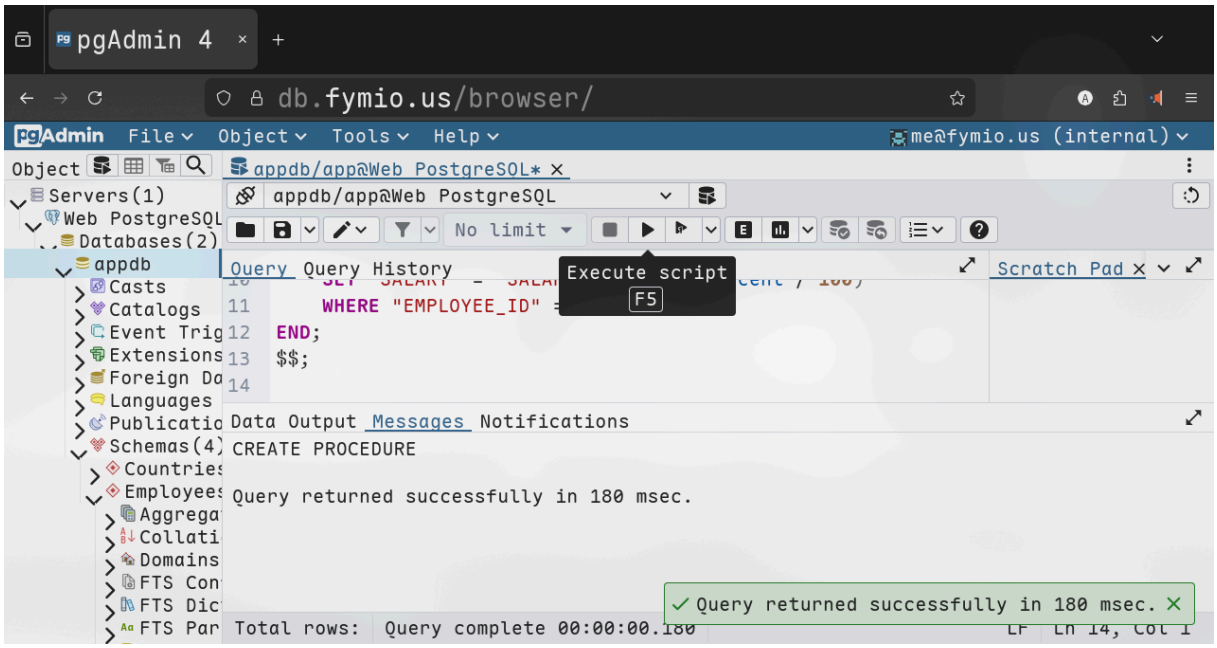

Рис. 19: Создание процедуры `raise_salary`.

Затем я проверил работу созданной хранимой процедуры. Для этого я проверил зарплату сотрудника с идентификатором `100`. (Рис. 20)

```
SELECT "SALARY" FROM "EmployeesDepartment"."EMPLOYEES" WHERE
"EMPLOYEE_ID" = 100;
```

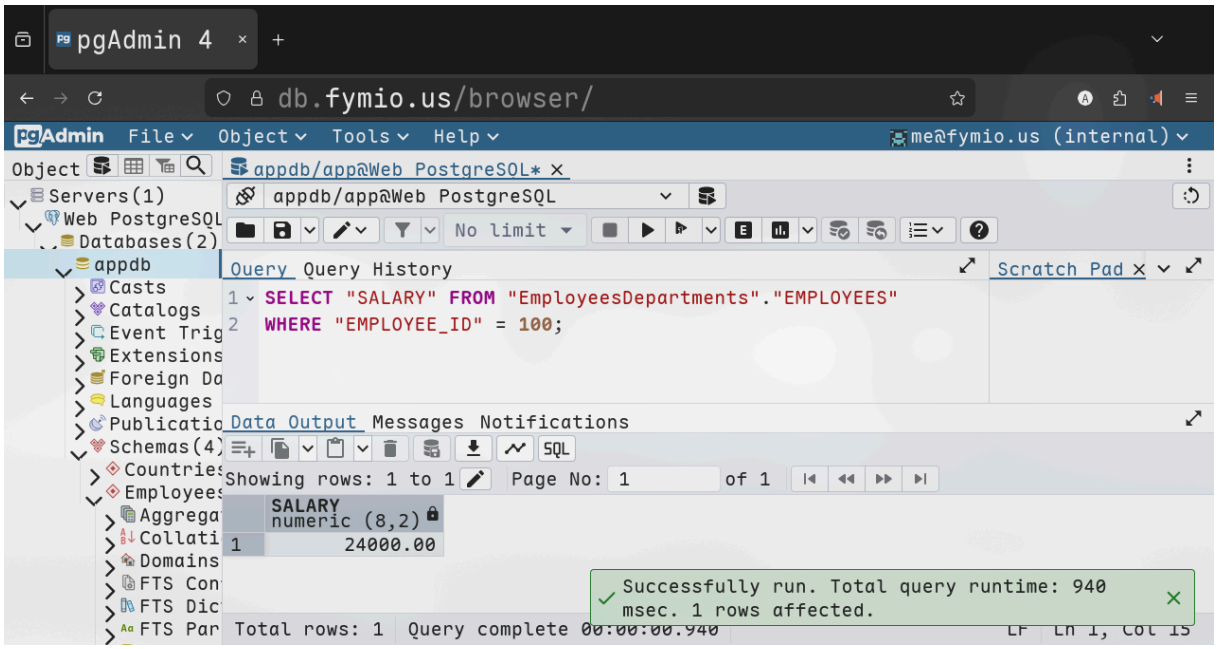

Рис. 20: Зарплата “номера 100”.

Затем я вызвал процедуру, которая повысила заработную плату сотруднику на 10%. (Рис. 21)

```
CALL "EmployeesDepartments".raise_salary(100, 10);
```

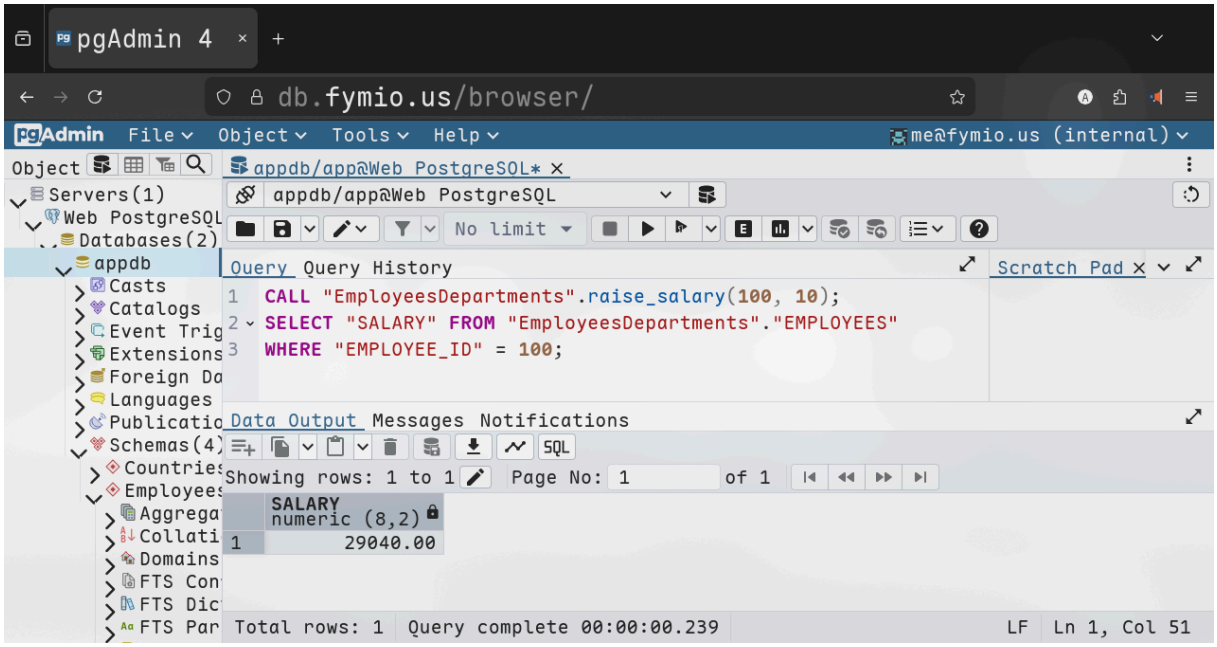

Рис. 21: Вызов процедуры `raise_salary`.

В процедуру не обязательно передавать параметры в той последовательности, в которой они были объявлены. К параметрам процедуры можно обращаться по имени. (Рис. 22)

```
SELECT * FROM "EmployeesDepartments"."EMPLOYEES" WHERE
"EMPLOYEE_ID" = 101;

CALL "EmployeesDepartments".raise_salary(p_percent:=10,
p_employee_id:=101);

SELECT * FROM "EmployeesDepartments"."EMPLOYEES" WHERE
"EMPLOYEE_ID" = 101;
```

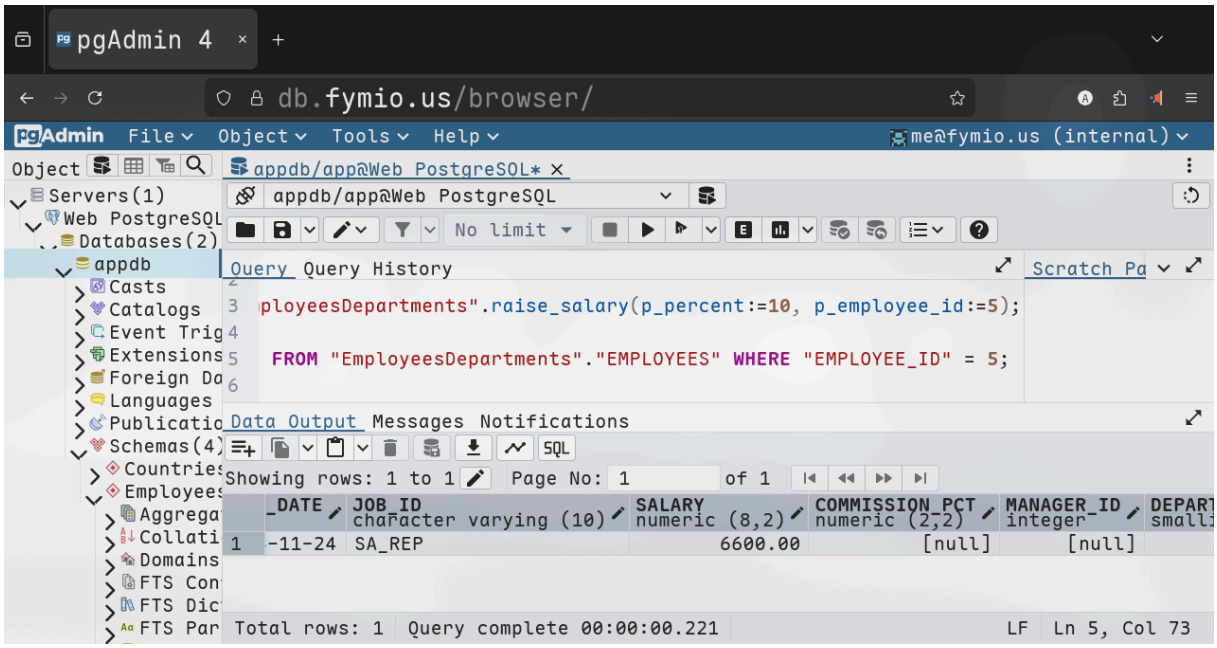

Рис. 22: Обращение к параметрам процедуры `raise_salary` по имени

Задание 7. Создание собственной хранимой процедуры

Я создал процедуру. (Рис. 23)

```
CREATE OR REPLACE PROCEDURE
"EmployeesDepartments".manage_employee(p_emp_id INT, p_mode
TEXT)
LANGUAGE plpgsql AS
$$
BEGIN
    IF NOT EXISTS(
        SELECT 1
        FROM "EmployeesDepartments"."EMPLOYEES"
        WHERE "EMPLOYEE_ID" = p_emp_id
    ) THEN
        RETURN;
    END IF;

    IF upper(p_mode) = 'INCREASE' THEN
        UPDATE "EmployeesDepartments"."EMPLOYEES"
        SET "SALARY" = round("SALARY" * 1.10)
        WHERE "EMPLOYEE_ID" = p_emp_id;
    ELSIF upper(p_mode) = 'BONUS' THEN
        UPDATE "EmployeesDepartments"."EMPLOYEES"
        SET "SALARY" = "SALARY" + 1000
        WHERE "EMPLOYEE_ID" = p_emp_id;
```

```
    ELSE
        RETURN;
    END IF;
END;
$$;
```

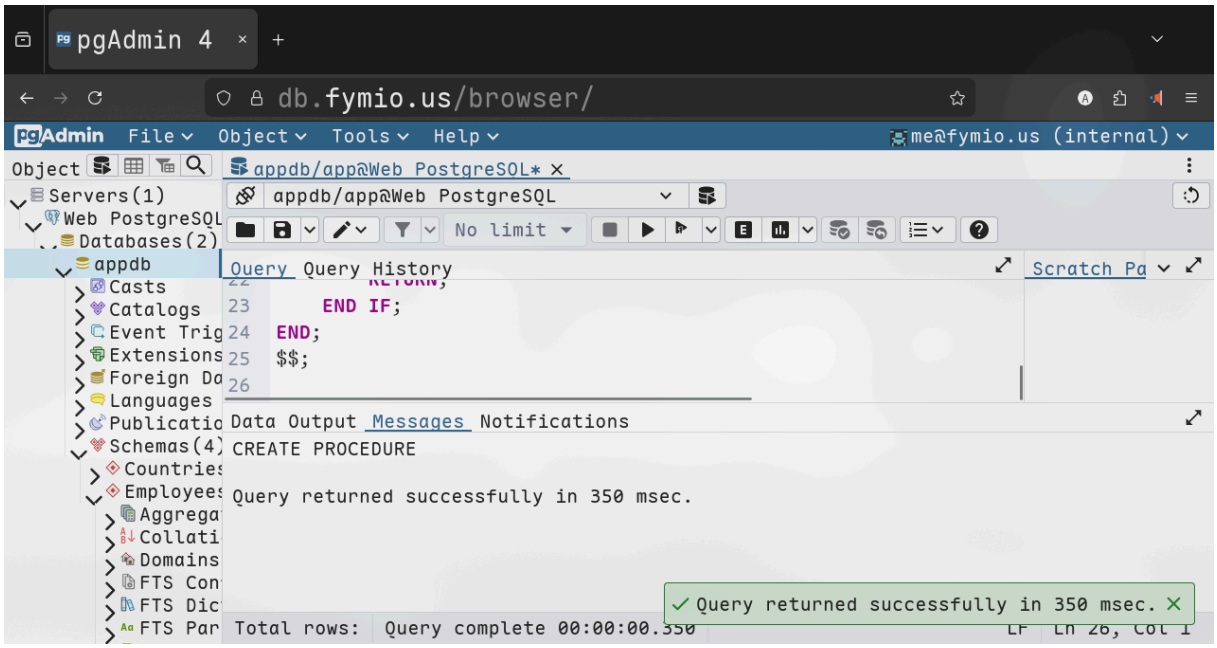

Рис. 23: Создание процедуры.

Режим `INCREASE`. Увеличение на 10% (Рис. 24)

```
SELECT "EMPLOYEE_ID", "FIRST_NAME", "LAST_NAME", "SALARY"
FROM "EmployeesDepartments"."EMPLOYEES"
WHERE "EMPLOYEE_ID" = 100;

CALL "EmployeesDepartments".manage_employee(100, 'INCREASE');

SELECT "EMPLOYEE_ID", "FIRST_NAME", "LAST_NAME", "SALARY"
FROM "EmployeesDepartments"."EMPLOYEES"
WHERE "EMPLOYEE_ID" = 100;
```

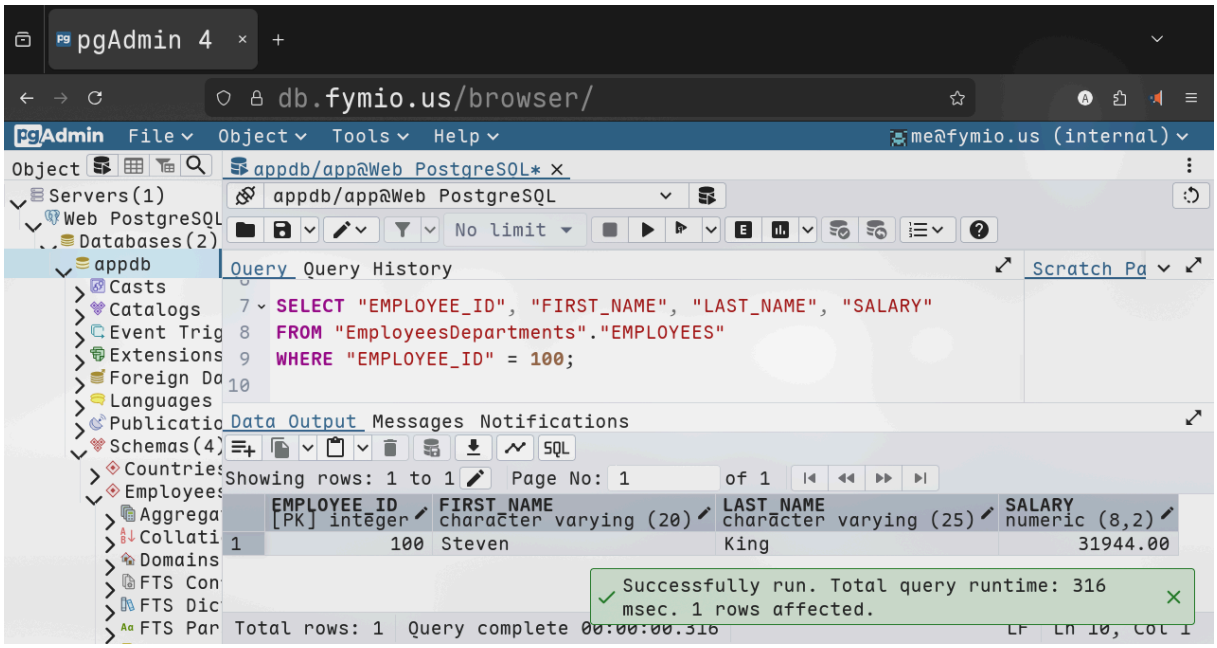

Рис. 24: Режим `INCREASE`.

Режим `BONUS`. Прибавление 1000 (Рис. 25)

```
SELECT "EMPLOYEE_ID", "FIRST_NAME", "LAST_NAME", "SALARY"
FROM "EmployeesDepartments"."EMPLOYEES"
WHERE "EMPLOYEE_ID" = 100;

CALL "EmployeesDepartments".manage_employee(100, 'BONUS');

SELECT "EMPLOYEE_ID", "FIRST_NAME", "LAST_NAME", "SALARY"
FROM "EmployeesDepartments"."EMPLOYEES"
WHERE "EMPLOYEE_ID" = 100;
```

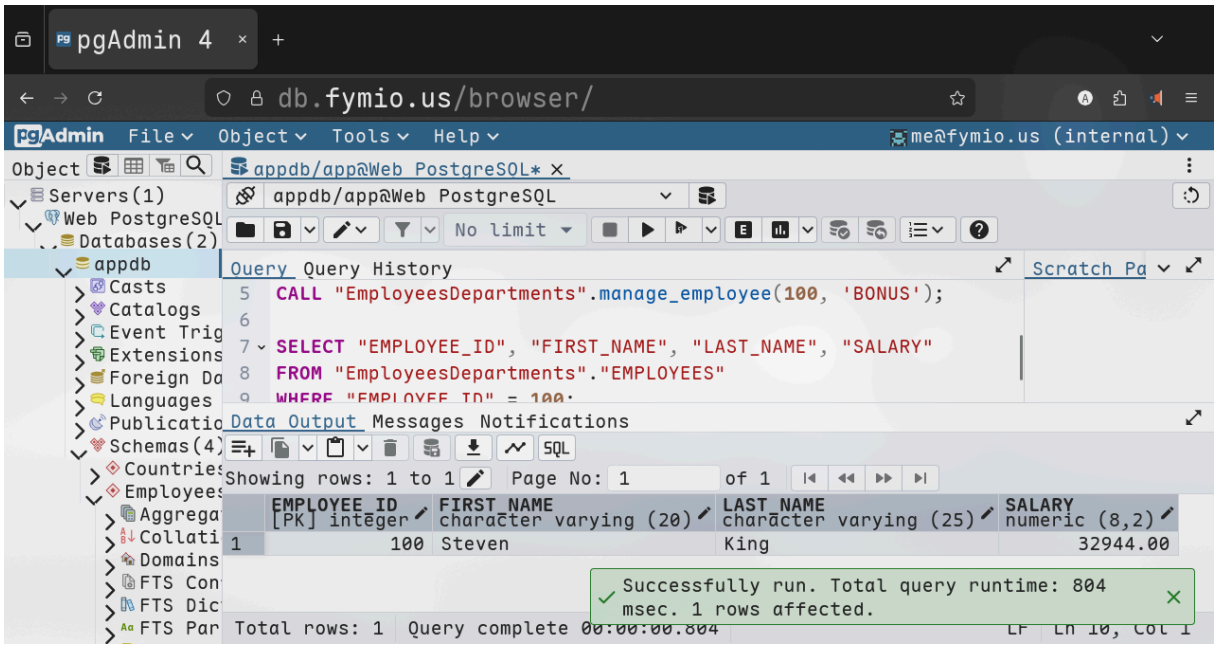

Рис. 25: Режим `BONUS`.

Выводы и анализ результатов работы

В ходе выполнения работы я последовательно создал обычные и материализованные представления, пользовательские функции и хранимые процедуры, а также проверил их работу на практических примерах. Была получена ясная разница между видами представлений: обычное всегда возвращает актуальные данные, так как вычисляет запрос при каждом обращении, а материализованное хранит результат и требует обновления, но работает быстрее. Это проявилось в том, что изменения в таблице “Employees” не отражались в материализованном представлении до выполнения команды обновления.

В процессе работы я освоил создание представлений как через интерфейс pgadmin, так и через sql, а также убедился, что изменения структуры таблиц, от которых зависят представления, могут быть заблокированы. Были созданы две скалярные функции — одна для получения полного имени сотрудника, другая для возврата зарплаты в разных режимах.

Также была реализована собственная процедура с ветвлением, выполняющая разные действия в зависимости от режима.