ПРАКТИЧЕСКАЯ РАБОТА 4.

Практическая работа №4 выполняется индивидуально по методическим указаниям и включает в себя несколько заданий.

По практической работе №4 формируется итоговый отчет, содержащий результат выполнения работы. Итоговый отчет должен содержать:

- Титульный лист
- Цель работы.
- Задачи, решаемые при выполнении работы.
- Исходные данные.
- Выполнение работы: Краткое описание процесса выполнения всех задач по шагам (при наличии нескольких шагов) со скриншотами.
- Выводы и анализ результатов работы. Обобщение результатов выполнения всех задач работы: чего должны были достичь, чего фактически достигли и каким образом, с какими трудностями столкнулись, какие проблемы на каких этапах выполнения возникли и как именно были решены.

Задание 1. Создание представления с помощью графического интерфейса

1. Откройте окно создания представления в БД HR в схеме "EmployeesDepartments" (рисунок 1.1):

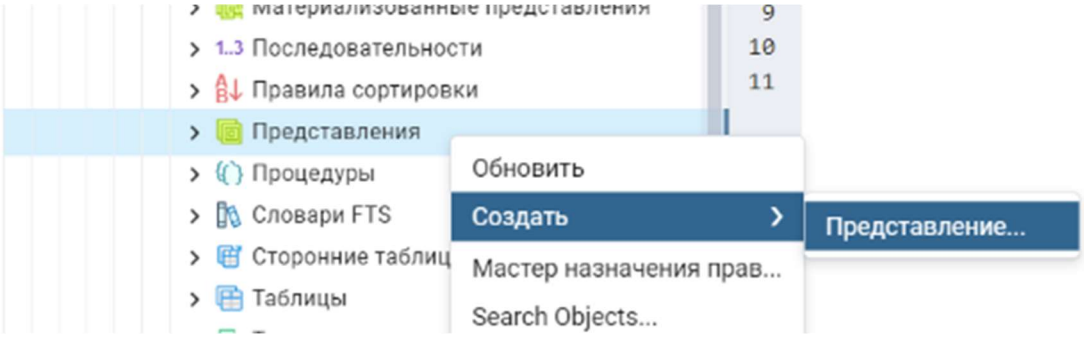

Рисунок 1.1 – Обозреватель объектов. Создание представления.

2. Введите наименование представления v_emp_active на вкладке General.
3. На вкладке «Код» введите его определение, как показано на рисунке 1.2.

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

Рисунок 1.2 – Создание тела представления v_emp_active.

Это представление позволяет отобрать только тех сотрудников, у которых заработная плата выше 5000. Предложение WHERE задаёт условие отбора сотрудников.

4. Изучите другие вкладки, особое внимание обратив на вкладку SQL (на ней отобразится полный текст DDL оператора CREATE) и нажмите «Сохранить».
5. Убедитесь в создании представления в обозревателе объектов и протестируйте его (рисунок 1.3).

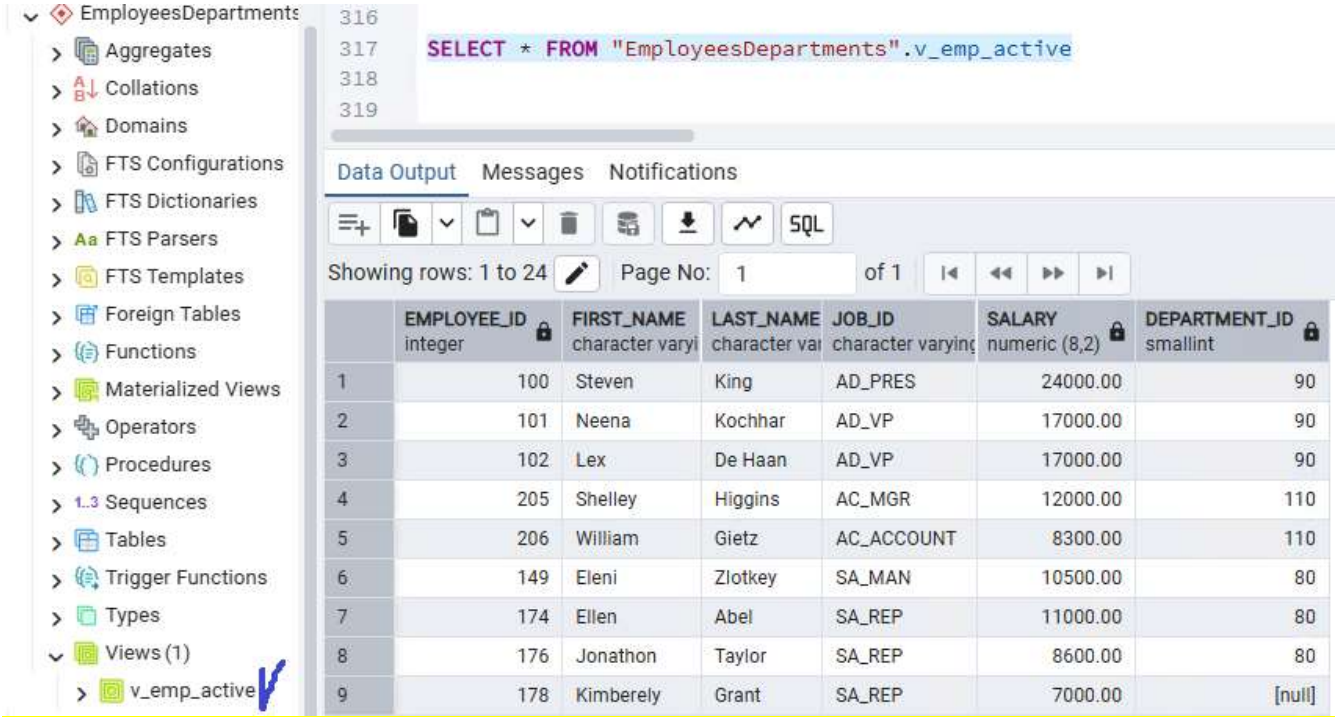

Рисунок 1.3 – Обозреватель объектов, представление v_emp_active

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

1. Создайте аналогичным образом в схеме "EmployeesDepartments" еще одно представление, которое имеет название V_EMP_ACTIVE_INFO и предназначено для отбора сотрудников из отдела 80 с заработной платой выше 5000. На этот раз в окне «Запросника» определите DDL оператор CREATE для создания нового представления (рисунок 2.1)

```
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;
```

Рисунок 2.1 – DDL оператор создания представления CREATE VIEW.

2. Протестируйте представление, результат должен соответствовать рисунку 2.2.

	EMPLOYEE_ID integer	full_name text	EMAIL character varying	JOB_ID character varying	SALARY numeric (8,2)	DEPARTMENT_ID smallint
1	149	Eleni Zlotkey	EZLOTKEY	SA_MAN	10500.00	80
2	174	Ellen Abel	EABEL	SA_REP	11000.00	80
3	176	Jonathon Tayl...	JTAYLOR	SA_REP	8600.00	80
4	212	Nick Hooper	NHOOPER	SR_SA_REP	9600.00	80

Рисунок 2.2 – Обращение к представлению V_EMP_ACTIVE_INFO.

3. Проведем дальнейшее тестирование представления "V_EMP_ACTIVE_INFO". Для этого добавим одну запись непосредственно в таблицу "EMPLOYEES", а вторую запись добавим через представление "v_emp_active" и проверим, что представление V_EMP_ACTIVE_INFO нам отображает корректные данные (рисунок 2.3).

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

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

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

Output Messages Notifications

SQL

ıg rows: 1 to 6 Page No: 1 of 1

EMPLOYEE integer	full_name text	EMAIL character varying	JOB_ID character varying	SALARY numeric (8,2	DEPARTI smallint
149	Eleni Zlotkey	EZLOTKEY	SA_MAN	10500.00	80
174	Ellen Abel	EABEL	SA_REP	11000.00	80
176	Jonathon Tayl...	JTAYLOR	SA_REP	8600.00	80
212	Nick Hooper	NHOOPER	SR_SA_REP	9600.00	80
239	Ivan Ivanov	Iv_Ivn@itmo.ru	SA_REP	6000.00	80
240	Petr Petrov	Pe_Ptr@itmo.ru	AD_VP	22000.00	80

Рисунок 2.3 – Проверка представления V_EMP_ACTIVE_INFO после добавления данных в таблицу.

PostgreSQL обычное представление (VIEW) является сохранённым SQL-запросом, который не хранит данные физически. При обращении к такому представлению система каждый раз выполняет исходный запрос к базовым таблицам и возвращает актуальные данные. Благодаря этому информация в представлении всегда соответствует текущему состоянию базы, но при сложных запросах это может снижать производительность, поскольку результат вычисляется заново при каждом обращении.

4. Теперь попытайтесь внести изменения в таблицу "EMPLOYEES" с помощью кода, представленного на рисунке 2.4.

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

Рисунок 2.4 – Изменение структуры базовой таблицы "EMPLOYEES".

Прочитайте текст ошибки, представленный на рисунке 2.5, и сделайте выводы.

```
ERROR:  cannot alter type of a column used by a view or rule
rule _RETURN on view "EmployeesDepartments".v_emp_active depends on column "FIRST_NAME"

SQL state: 0A000
Detail: rule _RETURN on view "EmployeesDepartments".v_emp_active depends on column "FIRST_NAME"
```

Рисунок 2.5 – Сообщение об ошибке при изменении базовой таблицы.

Задание 3. Создание материализованного представления

1. Материализованное представление (MATERIALIZED VIEW), в отличие от обычного, сохраняет результат выполнения запроса на диске как отдельный объект, похожий на таблицу. Это позволяет значительно ускорить доступ к данным, так как при запросах система обращается к уже сохранённым результатам, а не пересчитывает их.

   Пересоздайте существующее представление с использование директивы MATERIALIZED (рисунок 3.1).

Рисунок 3.1 – Создание материализованного представления.

2. Теперь данное представление должно работать быстрее, однако это незаметно на небольших данных. Протестируйте представление, выбрав данные сотрудника с фамилией «Ivanov», и зафиксируйте результат.
3. Обновите имя сотрудника с фамилией «Ivanov» в таблице "EMPLOYEES", установив ему имя «Petr» (рисунок 3.2).

```
UPDATE "EmployeesDepartments"."EMPLOYEES"
SET "FIRST_NAME" = 'Petr'
WHERE "LAST_NAME" = 'Ivanov';
```

Рисунок 3.2 – Изменение значения в таблице "EMPLOYEES".

Проверьте, что в таблице "EMPLOYEES" данные действительно изменились, и зафиксируйте результат изменения в отчете.

Однако материализованное представление не обновляется автоматически при изменении данных в исходных таблицах. Проверьте это и зафиксируйте в отчете.

4. Чтобы получить актуальную информацию из материализованного представления, его необходимо вручную обновить с помощью команды REFRESH MATERIALIZED VIEW (рисунок 3.3.). Проверьте, что после ручного обновления представления, данные действительно корректные при запросе из него.

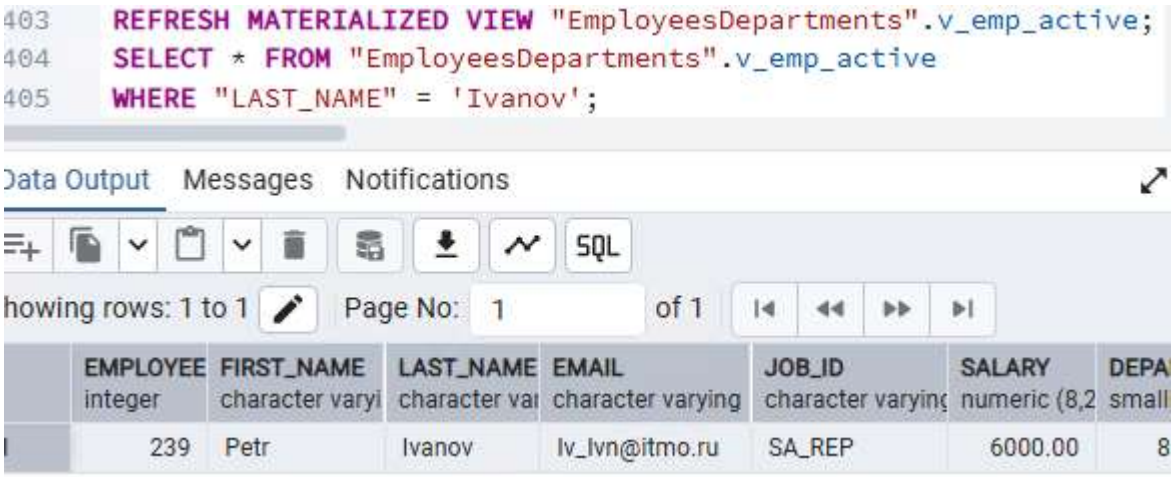

Рисунок 3.3 – Обновление материализованного представления.

Подумайте, какой можно сделать вывод, чем отличается материализованное представление от обычного? И в каких случаях лучше использовать каждый из этих типов.

Задание 4. Создание скалярной пользовательской функции

1. В окне Query Editor (Редактор запросов) создайте функцию (рисунок 4.1), 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 plpgsql;
```

Рисунок 4.1 – Создание скалярной пользовательской функции

Данная функция служит демонстрацией базового синтаксиса процедурного расширения SQL (PL/pgSQL) в СУБД Postgres. PL/pgSQL. Задача функции — по идентификатору сотрудника вернуть его полное имя, составленное из имени и фамилии.

Двойные знаки доллара $$ в PostgreSQL используются как специальные ограничители для обозначения начала и конца тела функции. Такой способ оформления называется *dollar-quoting*. Он позволяет помещать внутрь тела функции любой текст, включая одинарные кавычки и строковые литералы, без необходимости их экранирования. Это делает код более читаемым и снижает вероятность ошибок при написании многострочных конструкций.

Функция начинается с объявления имени и списка параметров: в данном случае используется один входной параметр p_emp_id, который должен иметь целый тип (INT), через который передаётся идентификатор сотрудника в функцию. Ключевое слово RETURNS определяет тип возвращаемого значения — в нашем случае это текстовая строка (TEXT).

После объявления функции идёт блок для объявления внутренних переменных функции, написанный на языке PL/pgSQL, этот блок начинается служебным словом DECLARE. В нашем случае объявляется переменная v_full_name, которая будет использована для хранения результата выборки. Ей определен тип данных TEXT.

Далее идёт основной блок функции – её тело. Оно ограничено служебными словами BEGIN……END. Именно здесь прописывается основная логика функции, в нашем случае она заключается в выполнении SQL-запроса, представленного на рисунке 4.2.

```
SELECT "FIRST_NAME"|| ' ' || "LAST_NAME"
INTO   v_full_name
FROM   "EmployeesDepartments"."EMPLOYEES"
WHERE  "EMPLOYEE_ID" = p_emp_id;
```

Рисунок 4.2 – Создание скалярной пользовательской функции

Конструкция SELECT ... INTO характерна для PL/pgSQL и служит для записи значения напрямую в переменную. Если сотрудник с указанным идентификатором существует, его имя и фамилия конкатенируются в одну строку и присваиваются переменной v_full_name. Если же запись не найдена, переменная остаётся равной NULL, что является корректным и ожидаемым поведением для данной функции.

Завершается тело функции оператором RETURN, который возвращает полученное значение вызывающей стороне. Поскольку PL/pgSQL автоматически прерывает выполнение после RETURN, дополнительная логика не требуется.

2. Протестируйте работоспособность скалярной функции, передав в нее известный вам идентификатор сотрудника. Пример вызова функции представлен на рисунке 4.3.

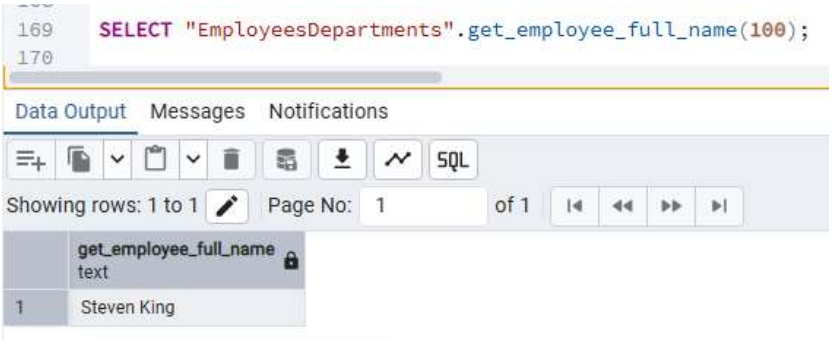

Рисунок 4.3 – Вызов пользовательской функции

Задание 5. Создание собственной скалярной функции

1. Создайте собственную скалярную функцию и опишите её назначение.

Требования к функции:

- Функция должна принимать два параметра. Первый параметр используется для отбора строк из выбранной таблицы (например, по EMPLOYEE_ID, DEPARTMENT_ID или другому признаку). Второй параметр задаёт режим работы функции (например: выполнять округление заработной платы или возвращать точное значение; учитывать комиссию или нет; приводить текст к верхнему/нижнему регистру и т.д.).
- Внутри функции должен быть реализован алгоритм с использованием оператора ветвления «IF…ELSE….END IF» или «CASE», если логику ветвления можно выразить в SQL-запросе.
- Если подходящие строки не найдены, функция должна возвращать.

2. Продемонстрируйте работоспособность функции:
- примеры вызова функции (как минимум два — по одному на каждый режим);
- зафиксируйте полученный результат выполнения;
- кратко прокомментируйте, почему получены именно такие значения.

Задание 6. Создание хранимой процедуры

1. В этом задании вы создадите хранимую процедуру, предназначенную для увеличения заработной платы выбранного сотрудника на указанный процент. Для этого напишите и выполните код, представленный на рисунке 6.1.

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

Рисунок 6.1 – Создание процедуры raise_salary

Создание процедуры обеспечивается DDL оператором CREATE (OR REPLACE) PROCEDURE в котором указывается имя процедуры «raise_salary» и список параметров в круглых скобках. В данном случае предусмотрено два входных параметра — p_employee_id и p_percent.

Параметр p_employee_id имеет тип INTEGER и используется для передачи идентификатора сотрудника, для которого выполняется повышение зарплаты.

Параметр p_percent имеет тип NUMERIC и задаёт процент увеличения заработной платы.

Так как процедура предназначена для выполнения действия (обновления данных) и не возвращает значения, в её объявлении отсутствует ключевое слово RETURN. Это отличает процедуру от функции, где всегда указывается тип возвращаемого результата.

Основная логика процедуры размещена между служебными словами BEGIN … END. Внутри данного блока выполняется оператор UPDATE, который изменяет значение поля "SALARY" в таблице "EMPLOYEES" схемы "EmployeesDepartments". Новое значение рассчитывается путём умножения текущей заработной платы на указанный процент. Таким образом, повышение осуществляется пропорционально переданному проценту.

Процедура не использует внутренние переменные, так как все необходимые значения передаются параметрами и сразу применяются в SQL-операции.

2. Проверьте работу созданной хранимой процедуры. Для этого сначала определите, какая зарплата у сотрудника с идентификатором 100. Запрос должен содержать информацию об идентификаторе, фамилии и заработной плате. Запрос и результат его выполнения зафиксируйте в отчете.

Вызовите процедуру, которая повышает заработную плату сотруднику на 10%. Передайте процедуре на вход параметры в том же самом порядке, что и при её объявлении (рисунок 6.2).

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

Рисунок 6.2 – Вызов процедуры raise_salary

Команда CALL используется в PostgreSQL для выполнения хранимой процедуры. В отличие от функций, которые вызываются через SELECT, процедуры запускаются именно с помощью CALL, потому что они предназначены для выполнения действий — обновления данных, работы с транзакциями или выполнения бизнес-логики.

Синтаксис вызова процедуры состоит из указания ключевого слова CALL за которым следует имя процедуры и список её параметров.

3. В процедуру не обязательно передавать параметры в той последовательности, в которой они были объявлены. К параметрам процедуры можно обращаться по имени, это продемонстрировано на рисунке 6.3.

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

Рисунок 6.3 – Обращение к параметрам процедуры raise_salary по имени

Проверьте, что в результате вызова хранимой процедуры значение заработной платы для сотрудника с идентификатором 101 изменилось. Зафиксируйте результат.

Задание 7. Создание собственной хранимой процедуры

1. Создайте собственную хранимую процедуру и опишите её назначение. Процедура должна выполнять одно из двух разных действий, в зависимости от режима работы.

Требования к процедуре:

- Процедура принимает два параметра (например, EMPLOYEE_ID, DEPARTMENT_ID или другой ID). Второй параметр задаёт режим работы процедуры, который определяет, какое действие будет выполнено.
- Внутри функции должен быть реализован алгоритм с использованием оператора ветвления «IF…ELSE….END IF».

- В зависимости от режима работы, процедура должна выполнять разные SQL-операции над таблицей, например:
  - ✓ повысить зарплату *или* назначить сотруднику комиссионный процент;
  - ✓ переместить сотрудника в другой отдел *или* изменить менеджера;
  - ✓ обновить название отдела *или* обновить его местоположение.
- Если подходящей строки не найдено, то процедура должна корректно завершиться (например, ничего не делать).

2. Продемонстрируйте работоспособность процедуры:
- примеры вызова процедуры в разных режимах;
- показать изменения в таблице до и после выполнения процедуры;
- описать, какое действие выполнила процедура в каждом вызове.