ПРАКТИЧЕСКАЯ РАБОТА №5. ОБЪЕКТЫ БАЗЫ ДАННЫХ: ПРЕДСТАВЛЕНИЯ И ХРАНИМЫЕ ПРОЦЕДУРЫ

Цель работы:

Работа направлена на формирование у студентов углубленных навыков работы с объектами баз данных в СУБД PostgreSQL, смещая акцент от прямого манипулирования данными к созданию переиспользуемых логических конструкций.

По завершении работы студент должен уметь:

- Сформировать практическое понимание представлений (Views) как механизма абстракции данных, упрощения сложных запросов и повышения уровня безопасности.
- Научиться чётко различать модифицируемые, немодифицируемые и материализованные представления, а также понимать конкретные условия, при которых представление в PostgreSQL становится обновляемым.
- Сформировать чёткое понимание различий между пользовательскими функциями (**Functions**) и хранимыми процедурами (**Stored Procedures**), а также сценариев их применения.
- Освоить создание пользовательских функций (**User-Defined Functions**) для инкапсуляции вычислений и возврата скалярных или табличных значений.
- Освоить создание хранимых процедур (**Stored Procedures**) для инкапсуляции бизнес-логики и выполнения последовательности **DML**-операций.
- Получить базовые навыки использования процедурного языка **PL/pgSQL**, включая объявление переменных, применение условной логики

(**IF...THEN...ELSE**) и генерацию пользовательских исключений (**RAISE EXCEPTION**).

- Развить умение корректно вызывать хранимые процедуры с помощью команды **CALL** и интерпретировать их результаты, включая как успешное завершение, так и обработку возвращаемых ошибок.

Постановка задачи:

Предварительное задание: в начале работы необходимо добавить скриншоты **всех используемых таблиц**.

Задание №1: создание модифицируемого представления

Для вашей базы данных создать **простое модифицируемое представление**, которое отбирает строки из одной таблицы по определенному критерию.

Например, для БД «Аптека» можно создать представление, отображающее лекарства только от одного производителя «Bayer AG»).

Задание №2: модификация данных через представление

Продемонстрировать возможность изменения данных в базовой таблице через представление, созданное в Задании №1. Для этого необходимо выполнить **два запроса**:

1. **Добавить** новую запись с помощью оператора **INSERT**.
2. **Удалить** существующую запись с помощью оператора **DELETE**.

Задание №3: создание немодифицируемого аналитического представления

Для вашей базы данных создать единое **немодифицируемое представление** для аналитических целей.

Представление должно объединять данные как минимум **из двух таблиц** и содержать **агрегирующие функции** (**COUNT**, **SUM**, **AVG** и т.д.) и группировку (**GROUP BY**).

Например, для БД «Аптека» такое представление могло бы для каждого производителя выводить общее количество наименований лекарств и их среднюю цену.

Задание №4: использование аналитического представления в запросах

Написать **SELECT**-запрос, который использует созданное в **Задании №3** **аналитическое представление** в качестве источника данных для дальнейшей фильтрации или анализа.

Например, можно отобрать производителей, у которых средняя цена на продукцию превышает определенное значение.

Задание №5: Создание и обновление материализованного представления

1. Создать **материализованное представление** для ускорения выполнения ресурсоемкого аналитического запроса.
   Например, для БД «Аптека» можно создать представление, которое заранее рассчитывает общую сумму продаж для каждого покупателя.
2. **Продемонстрировать** процесс обновления данных в представлении с помощью команды `REFRESH MATERIALIZED VIEW viewName;`.

Задание №6: разработка пользовательской функции для аналитических вычислений

1. **Разработать** пользовательскую функцию, которая инкапсулирует комплексный аналитический расчет. Функция должна принимать на вход идентификатор *(например, manufacturer_id)* и возвращать одно скалярное значение *(например, общую сумму продаж продукции данного*

производителя), вычисленное на основе соединения нескольких таблиц и применения агрегатных функций.

2. **Продемонстрировать** вызов функции в составе SELECT-запроса.

Задание №7: разработка хранимой процедуры для выполнения сложной операции

Разработайте хранимую процедуру, которая выполняет безопасную операцию по изменению данных. Процедура должна принимать на вход ID какой-либо записи и числовое значение *(например, количество)*.

Внутри процедуры необходимо проверить, достаточно ли текущего значения в числовом поле одной таблицы для выполнения операции.

- Если **да** – уменьшите это значение и добавьте новую запись в другую, связанную таблицу.
- Если **нет** – операция должна полностью прерваться, не внося никаких изменений в данные.

Для сообщения о результате используйте выходной параметр, который вернёт статус успеха или неудачи.

Задание №8: демонстрация вызова хранимой процедуры

Привести **два примера вызова процедуры**, созданной в Задании №7:

- **Успешный вызов**, который добавляет в вашу базу данных уникальную запись.
- **Неудачный вызов**, который демонстрирует срабатывание реализованной проверки целостности и возврат пользовательской ошибки.

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

ХОД ВЫПОЛНЕНИЯ РАБОТЫ

Введение

В предыдущих работах основной фокус был на извлечении и анализе данных с помощью сложных запросов, подзапросов и обобщенных табличных выражений.

Эта работа делает следующий шаг: мы научимся инкапсулировать (*прятать*) эту сложность внутри специальных объектов базы данных — представлений и хранимых процедур.

Это позволяет не только упростить код приложений, работающих с базой данных, но и перенести часть бизнес-логики непосредственно на уровень СУБД, повышая производительность, надежность и безопасность системы.

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

Таблица 3. Таблица sale items (Проданные лекарства)

	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 sales (Продажи)

	sale_id	customer_id	sale_date	total_amount
1	1	1	2025-01-22	221
2	2	2	2025-01-23	1 555,5

Таблица 5. Таблица customers (Покупатели)

	customer_id	first_name	last_name	email	phone_number
1	2	Петр	Петров	petr@example.com	+79004321567
2	1	Иван	Иванов	ivan@example.com	+79001234567

1. Работа с Представлениями (Views)

Представление (View) — это виртуальная таблица, основанная на сохраненном **SQL**-запросе. Для пользователя или приложения представление выглядит и ведет себя как обычная таблица, но на самом деле оно не хранит данные физически. Каждый раз, когда вы обращаетесь к представлению, СУБД выполняет лежащий в его основе **SELECT**-запрос и возвращает актуальный на данный момент результат.

Ключевые преимущества использования представлений:

- **Упрощение сложных запросов:** сложный запрос с множеством **JOIN** и вычислений можно «*спрятать*» в представление. После этого для получения данных достаточно будет выполнить простой `SELECT * FROM my_view;`
- **Безопасность:** можно предоставить пользователю доступ только к представлению, которое показывает разрешенные строки и столбцы *(например,* ***view_public_products****)*, скрывая конфиденциальную информацию *(например, закупочную цену)* из базовых таблиц.
- **Независимость от структуры:** структура базовых таблиц может меняться *(например, столбец* ***price*** *разделился на* ***base_price*** *и* ***tax****)*, но представление может сохранить прежний интерфейс для внешних приложений, вычисляя старый столбец **price** на лету.

1.1 Модифицируемые представления

Некоторые «*простые*» представления являются **модифицируемыми** *(обновляемыми)*.

Это означает, что через них можно выполнять **DML**-операции (**INSERT**, **UPDATE**, **DELETE**), и эти изменения будут автоматически применены **к базовой таблице**.

В **PostgreSQL** представление является автоматически модифицируемым, если оно удовлетворяет **всем следующим условиям**:

- В предложении **FROM** указана **ровно одна** базовая таблица *(или другое модифицируемое представление)*.
- Запрос **не содержит** *на верхнем уровне* предложений **WITH** (**CTE**), **DISTINCT**, **GROUP BY**, **HAVING**, **LIMIT** или **OFFSET**.
- Список выборки **SELECT** не содержит агрегатных или оконных функций, а также вычисляемых полей.
- Запрос не использует теоретико-множественные операции (**UNION**, **INTERSECT**, **EXCEPT**).

Листинг 1. Создание модифицируемого представления для лекарств компании «Bayer AG»

```
CREATE OR REPLACE VIEW bayer_medicines AS
SELECT
    id, name, quantity_in_stock, price,
    production_date, expiration_date, manufacturer_id
FROM
    medicines
WHERE
    manufacturer_id = 2;
```

Создаем представление, которое выбирает все столбцы из таблицы medicines, но только для производителя с ID=2 («Bayer AG»).

Такое представление является простым и полностью удовлетворяет правилам модифицируемого представления.

	id	name	quantity_in_stock	price	production_date	expiration_date	manufacturer_id
1	2	Аспирин	150	120	2025-07-12	2027-07-12	2
2	4	Витамин С	300	250	2025-07-10	2027-07-10	2

Рисунок 1 – Результат запроса

1.2 Модификация данных через представление

Через представление, созданное в предыдущем задании, можно добавлять и удалять записи, как если бы это была обычная таблица. СУБД автоматически применит эти изменения к базовой таблице medicines.

Листинг 2. Добавление данных через представление

```
INSERT INTO bayer_medicines (name, quantity_in_stock, price,
production_date, expiration_date, manufacturer_id)

VALUES ('Аспирин Кардио', 150, 180.00, '2025-08-01', '2028-08-01', 2);

SELECT * FROM medicines WHERE name = 'Аспирин Кардио';
```

Выполним добавление записи с помощью INSERT INTO, причём добавлять будем не сразу в исходную таблицу, а через представление.

Проверим, что запись действительно появилась в основной таблице medicines.

	id	name	quantity_in_stock	price	production_date	expiration_date	manufacturer_id
1	5	Аспирин Кардио	150	180	2025-08-01	2028-08-01	2

Рисунок 2 – Результат запроса – запись успешно добавлена

Теперь удалим это лекарство через то же самое представление, после чего проверим, что запись действительно была удалена из основной таблицы medicines.

Листинг 3. Удаление данных через представление

```
DELETE FROM bayer_medicines
WHERE name = 'Аспирин Кардио';

SELECT * FROM medicines WHERE name = 'Аспирин Кардио';
```

	id	name	quantity_in_stock	price	production_date	expiration_date	manufacturer_id

Рисунок 3 – Результат запроса – запись успешно удалена

1.3 Создание немодифицируемого аналитического представления

Если представление основано на запросе, который включает агрегатные функции (**COUNT**, **SUM**, **AVG**), группировку (**GROUP BY**) или соединение нескольких таблиц (**JOIN**), оно, как правило, становится немодифицируемым.

СУБД не может однозначно определить, как операция **INSERT** или **UPDATE** над одной строкой в представлении должна отразиться на множестве строк в базовых таблицах.

Такие представления предназначены только для чтения и идеально подходят для создания аналитических отчетов.

Листинг 4. Создание немодифицируемого аналитического представления

```
CREATE OR REPLACE VIEW manufacturer_summary AS
SELECT
    m.manufacturer_name,
    COUNT(med.id) AS total_medicines,
    COALESCE(ROUND(AVG(med.price), 2), 0) AS average_price,
    COUNT(med.id) FILTER (
        WHERE med.expiration_date BETWEEN CURRENT_DATE AND
              CURRENT_DATE + INTERVAL '30 day'
    ) AS expiring_soon_count
FROM
    manufacturers m
LEFT JOIN
    medicines med ON m.manufacturer_id = med.manufacturer_id
GROUP BY
    m.manufacturer_name;
```

В начале создаем представление, которое для каждого производителя выводит сводную информацию. Оно немодифицируемое, так как использует **JOIN**, **GROUP BY** и агрегатные функции.

Посчитаем общее количество наименований лекарств для данного производителя.

Далее вычисляем среднюю цену, округляя до 2 знаков после запятой. Используем **COALESCE**, чтобы в случае отсутствия лекарств у производителя вернуть 0, а не **NULL**.

Теперь посчитаем количество лекарств, срок годности которых истекает в ближайшие 30 дней. Для этого используем конструкцию **FILTER** (**WHERE**...), которая является более эффективным и читаемым способом условной агрегации в **PostgreSQL**.

Используем **LEFT JOIN**, чтобы в отчет попали даже те производители, у которых пока нет лекарств в нашей базе данных.

Получившийся результат группируем по производителю, чтобы агрегатные функции работали корректно для каждого из них.

	manufacturer_name	total_medicines	average_price	expiring_soon_count
1	ООО "Фармстандарт"	2	67,75	0
2	Bayer AG	2	185	0

Рисунок 4 – Результат запроса – аналитическое представление успешно создано

1.4 Использование аналитического представления в запросах

Представление, созданное ранее, инкапсулирует сложную логику.

Теперь для получения аналитической сводки или ее дальнейшей фильтрации достаточно выполнить простой запрос к самому представлению, как если бы это была обычная таблица.

Посмотрим, как это может выглядеть.

Листинг 5. Использование представления в качестве источника данных

```
SELECT
    manufacturer_name, average_price, total_medicines
FROM
    manufacturer_summary
WHERE
    average_price > 150.00
ORDER BY
    average_price DESC;
```

После создания представления manufacturer_summary, мы можем легко получить нужную информацию, не повторяя сложный запрос.

Например, найдем всех производителей, у которых средняя цена продукции превышает 150 рублей.

	manufacturer_name	average_price	total_medicines
1	Bayer AG	185	2

Рисунок 5 – Результат запроса – использование аналитического представления

1.5 Создание и обновление материализованного представления

Материализованное представление — это особый вид представления, который, в отличие от обычного, **физически хранит результат** своего запроса на диске. Это делает его похожим на **реальную таблицу**.

Основное преимущество: запросы к материализованному представлению выполняются **мгновенно**, так как СУБД не нужно каждый раз выполнять сложный базовый запрос.

Это идеальное решение для «тяжёлых» аналитических отчетов, которые не требуют данных в реальном времени.

Основной недостаток: данные в представлении могут **устаревать**. Для их обновления требуется специальная команда **REFRESH MATERIALIZED VIEW**.

Рассмотрим пример создания материализованного представления.

Листинг 6. Создание материализованного представления для отчета по продажам

```
CREATE MATERIALIZED VIEW TotalSalesByCustomer AS
SELECT
    c.customer_id,
    c.first_name,
    c.last_name,
    SUM(s.total_amount) AS total_spent
FROM
    customers c
JOIN
    sales s ON c.customer_id = s.customer_id
GROUP BY
    c.customer_id, c.first_name, c.last_name;

SELECT * FROM TotalSalesByCustomer;
```

После создания представления TotalSalesByCustomer, мы сможем, не повторяя ресурсоёмкий запрос, быстро получать результаты. Разумеется, в нашем случае он не слишком ресурсоёмкий, но в реальных проектах – весьма.

	customer_id	first_name	last_name	total_spent
1	1	Иван	Иванов	221

Рисунок 6 – Результат запроса – использование аналитического представления

Важно учитывать, что как было сказано выше, материальные представления не переформировываются при обычных запросах – они только возвращают данные.

При этом все остальные данные (в том числе те, которые послужили исходными для этого представления) вполне могут меняться, из-за чего со временем возникают несоответствия.

Предположим, что на следующий день после создания материализованного представления, 2 покупатель что-то купил.

	sale_id	customer_id	sale_date	total_amount
1	1	1	2025-01-22	221
2	2	2	2025-01-23	1 555,5

Рисунок 7 – Обновление данных таблицы Sales

Выполним запрос на обновление материализованного представления и проверим результаты.

Листинг 7. Обновление материализованного представления для отчета по продажам

```
REFRESH MATERIALIZED VIEW TotalSalesByCustomer;

SELECT * FROM TotalSalesByCustomer;
```

	customer_id	first_name	last_name	total_spent
1	1	Иван	Иванов	221
2	2	Петр	Петров	1 555,5

Рисунок 8 – Результат запроса – обновлённое аналитическое представление

Как видно, появилась вторая запись. Не забывайте об этом нюансе и регулярно обновляйте свои материализованные представления!

2. Пользовательские функции и хранимые процедуры

Хранимые процедуры и **пользовательские функции** — это именованные блоки кода на языке **PL/pgSQL**, которые хранятся и выполняются непосредственно на сервере базы данных.

Они позволяют инкапсулировать сложную бизнес-логику, повышая переиспользование кода, производительность и безопасность.

Ключевые различия: функция Vs процедура

Функция концептуально близка к **математической функции**: она принимает аргументы, выполняет вычисления и возвращает результат, **не изменяя** при этом состояние окружающего мира *(базы данных)*.

Процедура, напротив, представляет собой **последовательность действий** *(рецепт)*, которая целенаправленно изменяет состояние данных.

Ключевые отличия представлены в Таблице 1.

Таблица 1 – Ключевые отличия пользовательской функции от хранимой процедуры

Признак	Пользовательская функция	Хранимая процедура
Основное назначение	Выполнение вычислений, инкапсуляция сложной логики выборки данных.	Выполнение последовательности DML/DDL операций, инкапсуляция бизнес-процессов.
Возвращаемое значение	**Обязана** возвращать значение (скалярное или табличное) через **RETURN**.	**Не может** возвращать значение через **RETURN**. Результаты передаются через **OUT** или **INOUT** параметры.
Побочные эффекты (Side Effects)	Как правило, не должна изменять состояние БД (не содержит DML-операций **INSERT**, **UPDATE**, **DELETE**).	Создана для изменения состояния БД.
Управление транзакциями	Не может управлять транзакциями (COMMIT, ROLLBACK).	Может управлять транзакциями, что позволяет выполнять атомарные операции.
Способ вызова	Может быть вызвана как часть SELECT-запроса. SELECT get_revenue(1);	Вызывается отдельной командой CALL. CALL process_sale(1, 1, 10);

2.1 Разработка пользовательской функции

Это задание демонстрирует одно из главных преимуществ функций: инкапсуляцию сложного аналитического запроса.

Вместо того чтобы каждый раз писать громоздкий запрос с несколькими соединениями и агрегациями, можно «спрятать» эту логику в функцию и вызывать её по простому имени.

Это делает код приложения чище и снижает вероятность ошибок.

Листинг 8. Создание функции получения итогового дохода по производителю

```
CREATE OR REPLACE FUNCTION
get_manufacturer_total_revenue(p_manufacturer_id INT)
RETURNS DECIMAL(12, 2) AS $$
DECLARE
    total_revenue DECIMAL(12, 2);
    manufacturer_exists BOOLEAN;
BEGIN
    SELECT EXISTS(SELECT 1 FROM manufacturers
    WHERE id = p_manufacturer_id) INTO manufacturer_exists;

    IF NOT manufacturer_exists THEN
        RAISE EXCEPTION 'Manufacturer with ID % does not exist.',
                        p_manufacturer_id;
    END IF;

    SELECT
        COALESCE(SUM(si.quantity * si.unit_price), 0.00)
    INTO total_revenue
    FROM medicines AS med
    JOIN sale_items AS si ON med.id = si.medicine_id
    WHERE med.manufacturer_id = p_manufacturer_id;

    RETURN total_revenue;
END;
$$ LANGUAGE plpgsql;
```

В начале создаём (или заменяем) функцию, которая принимает ID производителя и возвращает общую выручку от продажи его продукции.

Указываем, что функция возвращает одно числовое значение.

Объявляем локальную переменную для хранения результата.

Выполняем **необходимые проверки** (здесь – на существование продавца).

Выполняем запрос с соединением таблиц medicines и sale_items для вычисления итоговой суммы, при этом используем **COALESCE** для возврата 0.00, если у производителя нет продаж – вместо NULL, что более удобно для отчётов.

С помощью оператора **INTO помещаем результат** запроса в нашу **локальную переменную**.

Возвращаем вычисленное значение с помощью **RETURN**.

ВАЖНО: в качестве результата должен быть приложен скриншот сообщения о создании процедуры, подобный **Рисунку 9**, представленному ниже.

Статистика 1

Name	Value
Updated Rows	0
Execute time	0.055s
Start time	Mon Oct 13 18:58:48 MSK 2025
Finish time	Mon Oct 13 18:58:48 MSK 2025
Query	CREATE OR REPLACE FUNCTION get_manufacturer_total_revenue(p_manufacturer_id INT)
	RETURNS DECIMAL(12, 2) AS $$
	DECLARE
	total_revenue DECIMAL(12, 2);
	manufacturer_exists BOOLEAN;
	BEGIN
	SELECT EXISTS(SELECT 1 FROM manufacturers
	WHERE id = p_manufacturer_id) INTO manufacturer_exists;
	IF NOT manufacturer_exists THEN
	RAISE EXCEPTION 'Manufacturer with ID % does not exist.',
	p_manufacturer_id;
	END IF;
	SELECT
	COALESCE(SUM(si.quantity * si.unit_price), 0.00)
	INTO total_revenue
	FROM medicines AS med
	JOIN sale_items AS si ON med.id = si.medicine_id
	WHERE med.manufacturer_id = p_manufacturer_id;
	RETURN total_revenue;
	END;
	$$ LANGUAGE plpgsql

Рисунок 9. Результат создания функции

2.2 Применение пользовательской функции

Продемонстрируем вызов функции для получения отчёта по выручке для всех производителей.

Обратите внимание: функция вызывается **для каждой строки** таблицы manufacturers.

Листинг 9. Вызов процедуры добавления лекарства с проверкой целостности

```
SELECT
    manufacturer_name,
    get_manufacturer_total_revenue(manufacturer_id::int) AS revenue
FROM
    manufacturers
ORDER BY
    revenue DESC;
```

	manufacturer_name	revenue
1	ООО "Фармстандарт"	10 506,5
2	Bayer AG	1 370

Рисунок 10 – Результат вызова пользовательской функции.

2.3 Создание хранимой процедуры

В отличие от функции, процедура идеально подходит для реализации многошаговых бизнес-операций, которые изменяют состояние базы данных.

В данном примере процедура process_medicine_sale инкапсулирует всю логику продажи:

- проверку остатков;
- списание товара;
- регистрацию продажи.

Режимы параметров: IN, OUT и INOUT

Для управления потоком данных в хранимых процедурах используются специальные режимы параметров, которые определяют, как процедура будет взаимодействовать с передаваемыми в неё переменными.

- **IN** – параметр для **входящих данных**. Этот параметр служит для передачи информации внутрь процедуры. Он работает по принципу «только для чтения»: процедура использует значение, но не может его изменить в исходном коде.

- **OUT** – параметр для **выходящих данных**. С помощью этого параметра процедура возвращает результат своей работы. Она берёт переменную как пустой контейнер, наполняет его данными и отдаёт обратно.

- **INOUT** – параметр для **входящих** и **выходящих данных**. Этот режим позволяет организовать полноценный обмен. Процедура получает переменную, считывает её текущее значение, вносит свои правки и возвращает её уже в обновлённом виде.

Создадим нашу процедуру продажи лекарства.

Листинг 10. Создание процедуры продажи лекарства

```
CREATE OR REPLACE PROCEDURE process_medicine_sale(
    p_medicine_id INT,       -- Входной параметр: ID лекарства
    p_customer_id INT,       -- Входной параметр: ID покупателя
    p_quantity_sold INT,     -- Входной параметр: продаваемое количество
    OUT p_success BOOLEAN,   -- Выходной параметр: флаг успеха операции
    OUT p_message TEXT       -- Выходной параметр: сообщение о результате
)
LANGUAGE plpgsql AS $$
DECLARE
    current_stock INT;
    current_price DECIMAL(10, 2);
    new_sale_id INT;
    customer_exists BOOLEAN; -- Объявляем переменную
    total_sale_amount DECIMAL(12, 2);
BEGIN
    -- Проверка существования покупателя
    SELECT EXISTS(SELECT 1 FROM customers WHERE customer_id = p_customer_id)
           INTO customer_exists;
    IF NOT customer_exists THEN
        p_success := FALSE;
        p_message := 'Ошибка: покупатель с ID ' || p_customer_id || ' не найден.';
        RETURN;
    END IF;

    -- Получаем данные о лекарстве и блокируем строку для безопасности
    SELECT quantity_in_stock, price
    INTO current_stock, current_price
    FROM medicines WHERE id = p_medicine_id FOR UPDATE;

    -- Проверка существования лекарства
    IF NOT FOUND THEN
        p_success := FALSE;
        p_message := 'Ошибка: лекарство с ID ' || p_medicine_id || ' не найден.';
        RETURN;
    END IF;

    IF current_stock >= p_quantity_sold THEN
        -- Уменьшаем остаток
        UPDATE medicines SET quantity_in_stock = quantity_in_stock - p_quantity_sold
        WHERE id = p_medicine_id;

        -- Рассчитываем итоговую сумму
        total_sale_amount := p_quantity_sold * current_price;

        -- Создаем новую продажу с корректной суммой
        INSERT INTO sales (customer_id, sale_date, total_amount)
        VALUES (p_customer_id, CURRENT_DATE, total_sale_amount)
        RETURNING sale_id INTO new_sale_id;

        -- Добавляем детализацию продажи
        INSERT INTO sale_items (sale_id, medicine_id, quantity, unit_price)
        VALUES (new_sale_id, p_medicine_id, p_quantity_sold, current_price);

        p_success := TRUE;
        p_message := 'Продано. Остаток: ' || (current_stock - p_quantity_sold);
    ELSE
        p_success := FALSE;
        p_message := 'Ошибка: недостаточно товара на складе. В наличии: ' ||
current_stock || ', запрашивается: ' || p_quantity_sold;
    END IF;
END;
$$;
```

Шаг 1: проверка наличия товара на складе.

FOR UPDATE блокирует строку на время транзакции, чтобы избежать «состояния гонки», когда два пользователя одновременно пытаются купить последний товар.

Шаг 2: условная логика для проверки, найдено ли лекарство. Если не найдено, то прерываем выполнение процедуры.

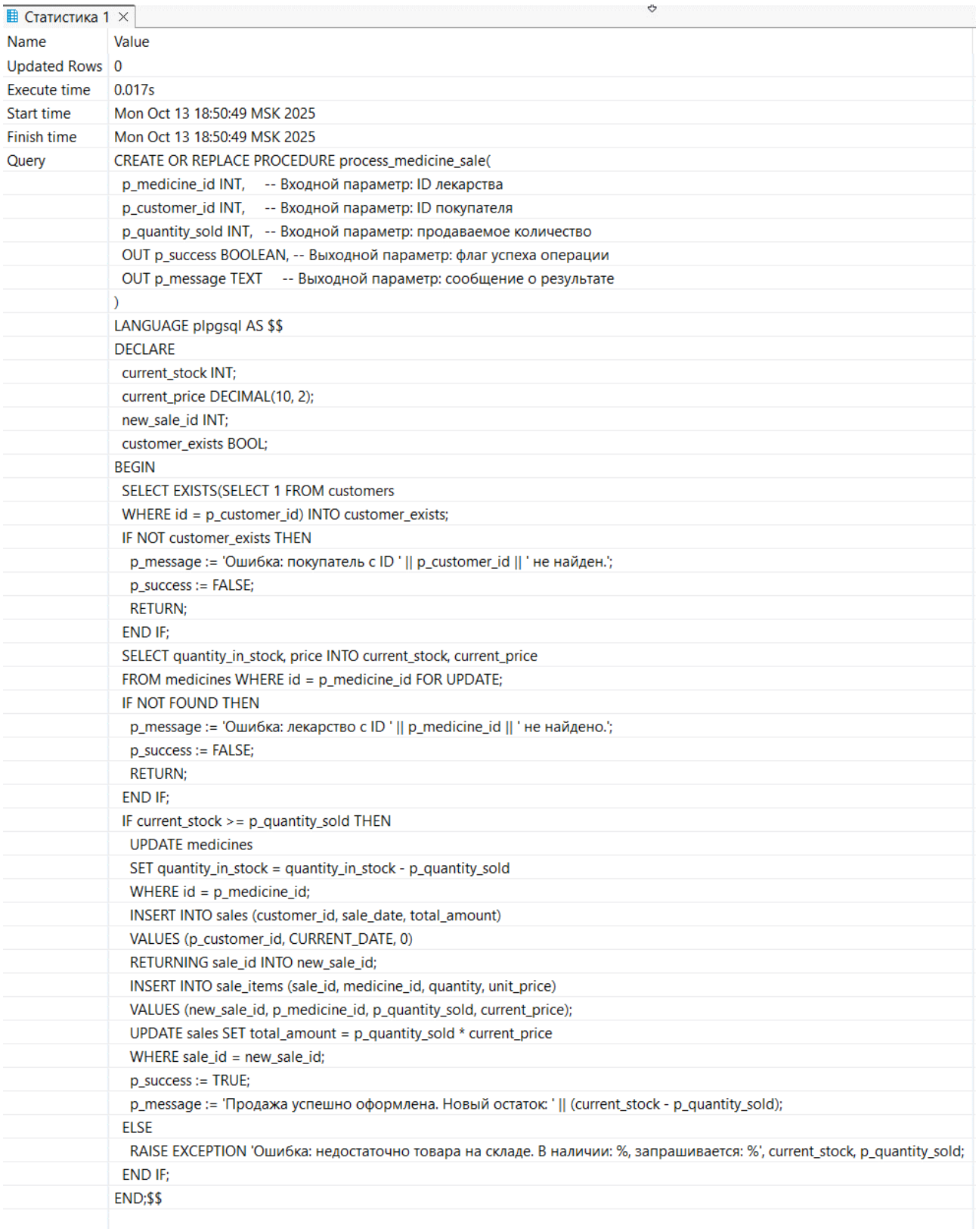
Шаг 3: проверяем, достаточно ли товара на складе.

Если товара достаточно:

- обновляем остатки в таблице medicines;
- регистрируем продажу в таблице sales (**RETURNING** sale_id позволяет сразу получить ID новой продажи);
- добавляем позицию в детализацию продажи (sale_items);
- обновляем общую сумму в чеке;
- устанавливаем выходные параметры для успешного случая.

Если товара недостаточно – генерируем **исключение** (ошибку).

ВАЖНО: в качестве результата должен быть приложен скриншот сообщения о создания процедуры, аналогичный **Рисунку 11**, представленному ниже.



Статистика 1

Name	Value
Updated Rows	0
Execute time	0.017s
Start time	Mon Oct 13 18:50:49 MSK 2025
Finish time	Mon Oct 13 18:50:49 MSK 2025
Query	CREATE OR REPLACE PROCEDURE process_medicine_sale(
	p_medicine_id INT, -- Входной параметр: ID лекарства
	p_customer_id INT, -- Входной параметр: ID покупателя
	p_quantity_sold INT, -- Входной параметр: продаваемое количество
	OUT p_success BOOLEAN, -- Выходной параметр: флаг успеха операции
	OUT p_message TEXT -- Выходной параметр: сообщение о результате
	)
	LANGUAGE plpgsql AS $$
	DECLARE
	current_stock INT;
	current_price DECIMAL(10, 2);
	new_sale_id INT;
	customer_exists BOOL;
	BEGIN
	SELECT EXISTS(SELECT 1 FROM customers
	WHERE id = p_customer_id) INTO customer_exists;
	IF NOT customer_exists THEN
	p_message := 'Ошибка: покупатель с ID ' || p_customer_id || ' не найден.';
	p_success := FALSE;
	RETURN;
	END IF;
	SELECT quantity_in_stock, price INTO current_stock, current_price
	FROM medicines WHERE id = p_medicine_id FOR UPDATE;
	IF NOT FOUND THEN
	p_message := 'Ошибка: лекарство с ID ' || p_medicine_id || ' не найдено.';
	p_success := FALSE;
	RETURN;
	END IF;
	IF current_stock >= p_quantity_sold THEN
	UPDATE medicines
	SET quantity_in_stock = quantity_in_stock - p_quantity_sold
	WHERE id = p_medicine_id;
	INSERT INTO sales (customer_id, sale_date, total_amount)
	VALUES (p_customer_id, CURRENT_DATE, 0)
	RETURNING sale_id INTO new_sale_id;
	INSERT INTO sale_items (sale_id, medicine_id, quantity, unit_price)
	VALUES (new_sale_id, p_medicine_id, p_quantity_sold, current_price);
	UPDATE sales SET total_amount = p_quantity_sold * current_price
	WHERE sale_id = new_sale_id;
	p_success := TRUE;
	p_message := 'Продажа успешно оформлена. Новый остаток: ' || (current_stock - p_quantity_sold);
	ELSE
	RAISE EXCEPTION 'Ошибка: недостаточно товара на складе. В наличии: %, запрашивается: %', current_stock, p_quantity_sold;
	END IF;
	END;$$

Рисунок 11 – Результат запроса

2.4 Демонстрация вызова хранимой процедуры

Для выполнения хранимой процедуры используется команда **CALL**.

Продемонстрируем два сценария:

- успешное добавление;
- попытку добавления дубликата, при которой сработает наша проверка.

Для начала проверим, какое количество лекарства доступно в данный момент.

Листинг 11. Проверка текущего количества

```
SELECT name, quantity_in_stock FROM medicines WHERE id = 1;
```

Результаты запроса видно на Рисунке 9.

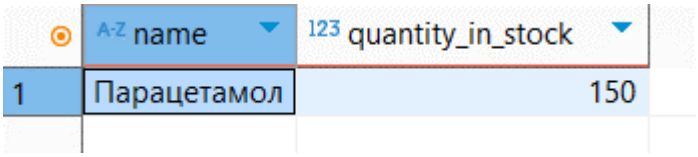


Рисунок 12 – Результат проверки

Теперь создадим запрос для успешной продажи лекарства с помощью процедуры.

Листинг 13. Продажа лекарства с помощью процедуры

```
DO $$
DECLARE
    v_success BOOLEAN;
    v_message TEXT;
BEGIN
    CALL process_medicine_sale(1, 1, 5, v_success, v_message);
    -- Выводим сообщение от процедуры в консоль.
    RAISE NOTICE '%', v_message;
END;
$$;
```

В начале создадим с помощью DECLARE необходимые переменные, указав их тип.

Далее вызываем функцию, передавая ей необходимые параметры, а также созданные переменные (для получения данных о результатах).

По окончании выполнения выводим в консоль сообщение с результатом.

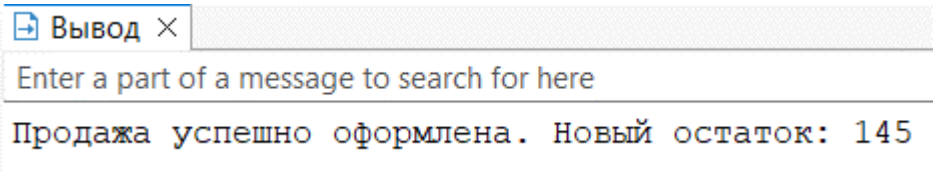


Рисунок 14 – Успешная продажа с использованием процедуры

Повторно вызываем процедуру, передавая ей те же самые аргументы, что и в первый раз, до тех пор, пока все товары не будут распроданы.

После этого выполним процедуру снова, и увидим, что на этот раз возвращается ошибка.

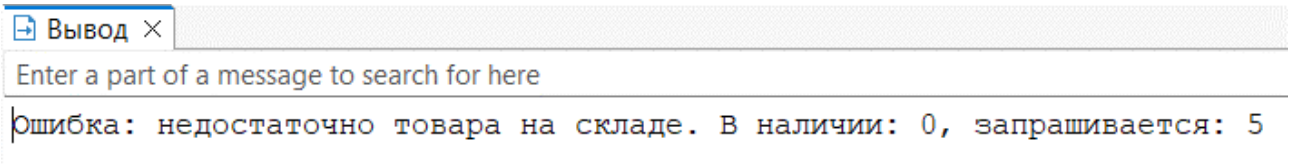


Рисунок 15 – Ошибка добавления новой записи – с нашим текстом

Выполнение процедуры было прервано, и PostgreSQL вернул **нашу** ошибку с текстом, который был указан в **RAISE EXCEPTION**.

КОНТРОЛЬНЫЕ ВОПРОСЫ

1. В чём заключается ключевое различие между обычным и материализованным представлением с точки зрения хранения данных и их актуальности? В какой ситуации вы бы выбрали для использования материализованное представление и почему?

2. Объясните, почему представление, содержащее агрегатную функцию (например, **SUM** или **COUNT**), не является автоматически модифицируемым в PostgreSQL.

3. Объясните принципиальную разницу в назначении между Представлением (**View**) и Хранимой процедурой (**Stored Procedure**). В какой ситуации для решения задачи вы бы выбрали представление, а в какой — процедуру?

4. Каково назначение режимов параметров **IN**, **OUT** и **INOUT** в хранимых процедурах? Приведите гипотетический пример задачи, для решения которой мог бы понадобиться параметр с режимом **OUT**.

5. В чём преимущество использования хранимой процедуры для оформления продажи (Задание 7) по сравнению с выполнением отдельных SQL-запросов (UPDATE, INSERT)?

КРАТКИЙ СПРАВОЧНЫЙ МАТЕРИАЛ

1. Представления (Views)

1.1 Что такое оконные функции и зачем они нужны?

Представления являются мощным инструментом для абстрагирования и упрощения работы с данными. В PostgreSQL их можно условно разделить на три типа, каждый из которых имеет свои особенности и сценарии использования.

Таблица 2. Сравнение типов представлений в PostgreSQL

Тип представления	Описание	Хранение данных	Обновление данных	Основной сценарий использования
Стандартное (*View*)	Виртуальная таблица, основанная на SQL-запросе.	Не хранит данные, запрос выполняется при каждом обращении.	Динамическое, всегда отражает актуальное состояние базовых таблиц.	Упрощение сложных запросов, контроль доступа к данным (безопасность).
Модифицируемое	Стандартное представление, удовлетворяющее набору строгих правил (одна таблица в **FROM**, нет агрегаций, **GROUP BY** и т.д.).	Не хранит данные.	Позволяет выполнять DML-операции (**INSERT**, **UPDATE**, **DELETE**), которые транслируются в базовую таблицу.	Предоставление безопасного и простого интерфейса для изменения подмножества данных.
Материализованное	Физическая копия результата запроса, которая хранится на диске.	Хранит данные физически, как обычная таблица.	Требует ручного обновления командой **REFRESH MATERIALIZED VIEW**. Данные могут быть неактуальными.	Ускорение выполнения сложных и ресурсоемких аналитических запросов, которые не требуют данных в реальном времени.

2. Хранимые Процедуры и PL/pgSQL

Хранимая процедура — это именованный блок кода на языке PL/pgSQL, предназначенный для выполнения последовательности действий и инкапсуляции бизнес-логики на сервере базы данных.

В отличие от представлений, которые служат для упрощения чтения данных, процедуры предназначены для их модификации и выполнения сложных операций.

Листинг 14. Структура блока PL/pgSQL

```
CREATE PROCEDURE...
LANGUAGE plpgsql
AS $$
DECLARE
    -- Здесь объявляются локальные переменные
    my_variable INT := 10;
BEGIN
    -- Здесь размещается исполняемый код (логика процедуры)
EXCEPTION
    -- Здесь можно обрабатывать исключения (ошибки)
    WHEN OTHERS THEN
        RAISE NOTICE 'Произошла ошибка!';
END;
$$;
```

Параметры процедуры

- **IN**: входной параметр *(значение передается в процедуру)*. Является режимом по умолчанию.
- **OUT**: выходной параметр *(значение присваивается внутри процедуры и возвращается вызывающей стороне)*.
- **INOUT**: входной и выходной параметр *(значение передается в процедуру, может быть изменено и возвращено)*.

Вызов процедуры

Для выполнения хранимой процедуры используется команда **CALL**:

CALL <*имя_процедуры*>(*параметры*);

Основные операторы управления потоком

Условный оператор IF:

Листинг 15. Условный оператор IF

```
IF <условие> THEN
    -- код, если условие истинно
ELSIF <другое_условие> THEN
    -- код, если другое условие истинно
ELSE
    -- код, если все условия ложны
END IF;
```

Генерация исключения RAISE:

Листинг 16. Прерывание выполнение с возвратом ошибки с предзаданным сообщением.

```
-- % - это место для подстановки значения переменной.

RAISE EXCEPTION 'Сообщение об ошибке: %', переменная;
```