



МИНОБРНАУКИ РОССИИ

Федеральное государственное бюджетное образовательное учреждение
высшего образования
«МИРЭА – Российский технологический университет»

РТУ МИРЭА

**Институт информационных технологий (ИИТ)
Кафедра цифровой трансформации (ЦТ)**

ОТЧЕТ ПО ПРАКТИЧЕСКОЙ РАБОТЕ № 7
по дисциплине «Разработка баз данных»

Студент группы *ИНБО-12-23. Албахтин И.В.*

(подпись)

Ассистент *Брайловский А.В.*

(подпись)

Москва 2025 г.

ПРАКТИЧЕСКАЯ РАБОТА №7. ОПТИМИЗАЦИЯ ЗАПРОСОВ И УПРАВЛЕНИЕ ТРАНЗАКЦИЯМИ В POSTGRESQL

Цель работы:

Целью данной практической работы является формирование у студентов практических навыков анализа и оптимизации производительности SQL-запросов, а также освоение механизмов управления транзакциями для обеспечения целостности данных (согласно принципам ACID) в СУБД PostgreSQL.

По завершении работы студент должен уметь:

- Сформировать практический навык анализа производительности SQL-запросов с использованием инструмента EXPLAIN ANALYZE.
- Научиться интерпретировать планы выполнения (ПВ), выявляя неэффективные операции, такие как полное сканирование таблицы (Seq Scan).
- Освоить создание различных типов индексов (B-tree, Partial, Function-based), как основного средства для ускорения операций поиска данных.
- Закрепить понимание транзакций как логической единицы работы и освоить использование команд BEGIN, COMMIT и ROLLBACK для обеспечения атомарности операций.
- Научиться моделировать и устранять проблемы параллельного доступа (аномалию «Неповторяемое чтение») с помощью уровней изоляции транзакций (REPEATABLE READ).

Постановка задачи:

Для выполнения практической работы необходимо последовательно выполнить следующие шаги, адаптируя примеры из БД «Аптека» к вашей собственной базе данных:

Подготовка базы данных.

Следуя руководству, наполнить одну из ключевых таблиц вашей БД большим объемом данных (не менее 20 000 строк). Это обязательно для корректной демонстрации работы оптимизатора.

Задание №1: анализ и оптимизация (3 сценария)

Определить три различных «медленных» запроса к вашей БД, которые можно оптимизировать с помощью разных типов индексов (например, стандартный B-Tree, индекс по выражению, частичный/отфильтрованный индекс).

Если в вашей базе недостаточно данных, выполните для одной из своих таблиц действия по автоматической генерации содержимого, описанные в разделе «Подготовка базы данных».

Для каждого из 3-х сценариев:

1. Выполнить анализ запроса «КАК ЕСТЬ» (без индекса) с помощью EXPLAIN ANALYZE.
2. Привести план выполнения «ДО», письменно проанализировать его и выявить причину низкой производительности (например, Seq Scan).
3. Создать необходимый INDEX для оптимизации.
4. Повторно выполнить EXPLAIN ANALYZE.

5. Привести план выполнения «ПОСЛЕ», демонстрирующий использование индекса (например, Index Scan).

6. Обязательно сформировать сравнительную таблицу (см. Таблица 1), демонстрирующую разницу в производительности (план, Execution Time) «ДО» и «ПОСЛЕ».

Задание №2: демонстрация атомарной транзакции (COMMIT).

По примеру раздела 2.2 реализуйте в своей базе одну бизнес-операцию (минимум две связанные операции изменения данных) внутри транзакции BEGIN...COMMIT.

Задокументировать все шаги и результаты, сделать выводы.

Задание №3: демонстрация отката транзакции (ROLLBACK). Адаптировать приведённый в разделе 2.3 SQL-скрипт, моделирующий сбой операции, под свою предметную область, повторив описанные действия.

Задокументировать все шаги и результаты, сделать выводы.

Задание №4: моделирование аномалии «Неповторяющееся чтение».

Используя два редактора SQL, смоделировать проблему «неповторяющееся чтение» на уровне изоляции по умолчанию (READ COMMITTED) по приведённому в разделе 2.4 образцу.

Задокументировать все шаги и результаты, сделать выводы.

Задание №5: устранение аномалии «Неповторяющееся чтение».

Повторить моделирование из Задания №4, но с использованием уровня изоляции REPEATABLE READ. В качестве образца использовать раздел 2.5.

Задокументировать все шаги и результаты, сделать выводы.

ВЫПОЛНЕНИЕ ПРАКТИЧЕСКОЙ РАБОТЫ

Таблица 1. Структура таблицы invoice

```
④ SELECT column_name, data_type, is_nullable  
      FROM information_schema.columns  
     WHERE table_name = 'invoice'  
     ORDER BY ordinal_position;
```

columns 1 X

ELECT column_name, data_type, is_nullable | Введите SQL выражение чтобы отфи

	A-Z column_name	A-Z data_type	A-Z is_nullable
1	invoice_id	integer	NO
2	maintenance_id	integer	YES
3	total_amount	numeric	YES
4	payment_status	character varying	YES

Таблица 2. Структура таблицы maintenance_work

```
④ SELECT column_name, data_type, is_nullable  
      FROM information_schema.columns  
     WHERE table_name = 'maintenance_work'  
     ORDER BY ordinal_position;
```

columns 1 X

ELECT column_name, data_type, is_nullable | Введите SQL выражение чтобы отфи

	A-Z column_name	A-Z data_type	A-Z is_nullable
	maintenance_work	intger	NO
	maintenance_id	integer	YES
	work_type_id	integer	YES
	part_id	integer	YES
	quantity	integer	YES

Таблица 3. Таблица invoice

The screenshot shows the MySQL Workbench interface with the 'invoice' table selected. The table has six rows of data with the following columns: invoice_id, maintenance_id, total_amount, and payment_status.

	invoice_id	maintenance_id	total_amount	payment_status
1	3	31	35 000	Оплачено
2	5	33	26 000	Оплачено
3	6	34	40 000	Оплачено
4	7	35	15 000	Оплачено
5	8	36	17 000	Оплачено
6	9	37	22 000	Ожидает оплаты

Таблица 4. Таблица maintenance_work

The screenshot shows the MySQL Workbench interface with the 'maintenance_work' table selected. The table has two rows of data with the following columns: maintenance_id, work_type_id, and part_id.

	maintenance_id	work_type_id	part_id
1	1	31	[NULL]
2	2	32	[NULL]

Задание №1: анализ и оптимизация (3 сценария)

The screenshot shows the dbstud application interface. On the left is a vertical toolbar with icons for file operations, AI, and other database management functions. The main area contains a script editor with the following PL/pgSQL code:

```
DO $$  
DECLARE  
    max_maint INT;  
BEGIN  
    SELECT MAX(maintenance_id)  
    INTO max_maint  
    FROM maintenance_work;  
    FOR i IN 1..20000 LOOP  
        INSERT INTO invoice (  
            maintenance_id,  
            total_amount,  
            payment_status  
        )  
        VALUES (  
            (1 + floor(random() * max_maint))::INT,  
            (1000 + random() * 50000)::NUMERIC(10,2),  
            CASE floor(random() * 2)  
                WHEN 0 THEN 'Ожидает оплаты'  
                ELSE 'Оплачено'  
            END  
        );  
    END LOOP;  
END;  
$$ LANGUAGE plpgsql;
```

Below the script editor is a statistics window titled "Статистика 1" (Statistics 1) containing the following data:

Name	Value
Updated Rows	0
Execute time	0,451s
Start time	Fri Dec 05 19:58:48 MSK 2025
Finish time	Fri Dec 05 19:58:49 MSK 2025
Query	DO \$\$ DECLARE max_maint INT; BEGIN SELECT MAX(maintenance_id) INTO max_maint FROM maintenance_work; FOR i IN 1..20000 LOOP INSERT INTO invoice (maintenance_id, total_amount, payment_status) VALUES ((1 + floor(random() * max_maint))::INT, (1000 + random() * 50000)::NUMERIC(10,2), CASE floor(random() * 2) WHEN 0 THEN 'Ожидает оплаты' ELSE 'Оплачено' END); END LOOP; END; \$\$ LANGUAGE plpgsql;

Рисунок 1 – Скрипт заполнения 20 000 случайными валидными записями (случайно запустил 2 раза)

The screenshot shows the dbstud application interface. In the top navigation bar, there are tabs for 'albakhtin_iv' (selected), '*<dbstud> Script' (with a green checkmark icon), and '123 invoice_id'. Below the tabs is a toolbar with icons for file operations: back, forward, new, open, save, print, and others. A vertical sidebar on the left contains icons for 'AI' and 'File'. The main area displays a query results window titled 'Результат 1'. The query 'SELECT COUNT(*) AS invoice_count FROM invoice;' is shown in the results pane, which displays a single row of data: '1' in the first column and '40 014' in the second column, under the heading '123 invoice_count'. The status bar at the bottom right says 'Введите SQL выражение'.

Рисунок 2 – Проверка количества строк в таблице invoice после генерации тестовых данных

```
⊖ SELECT COUNT(*) AS invoice_count
  FROM invoice;

⊖ EXPLAIN ANALYZE
SELECT *
  FROM invoice
 WHERE total_amount > 30000;
```

Результат 1 ×

PLAIN ANALYZE SELECT * FROM invoice WHERE total_amount > 30000;

A-Z QUERY PLAN

Seq Scan on invoice (cost=0.00..852.17 rows=1684)

Filter: (total_amount > '30000'::numeric)

Rows Removed by Filter: 23230

Planning Time: 0.406 ms

Execution Time: 15.122 ms

Рисунок 3 – План выполнения запроса до оптимизации

```
WHERE total_amount > 30000,
```

```
⊖ CREATE INDEX idx_invoice_total_amount
  ON invoice(total_amount);
```

Статистика 1 ×

	Value
Affected Rows	0
Exe time	0,065s
Time	Fri Dec 05 20:06:29 MSK 2025
Time	Fri Dec 05 20:06:29 MSK 2025
CREATE INDEX idx_invoice_total_amount	
ON invoice(total_amount)	

Рисунок 4 – Создание индекса по total_amount

```

EXPLAIN ANALYZE
SELECT *
FROM invoice
WHERE total_amount > 45000;

```

результат 1 ×

EXPLAIN ANALYZE SELECT * FROM invoice W | Введите SQL выражение чтобы отфильтровать результаты

A-Z QUERY PLAN

1	Bitmap Heap Scan on invoice (cost=108.44..518.74 rows=4664 width=38) (actual time=1.315..8.208 rows=4664)
2	Recheck Cond: (total_amount > '45000'::numeric)
3	Heap Blocks: exact=352
4	-> Bitmap Index Scan on idx_invoice_total_amount (cost=0.00..107.27 rows=4664 width=0) (actual time=1.207)
5	Index Cond: (total_amount > '45000'::numeric)
6	Planning Time: 0.385 ms
7	Execution Time: 8.482 ms

Рисунок 5 – План выполнения запроса после оптимизации

Метрика	До оптимизации	После оптимизации	Вывод
План выполнения	Seq Scan	Bitmap Index Scan + Bitmap Heap Scan	Индекс используется
Execution Time	~15 ms	~8 ms	Запрос ускорен примерно в 2 раза
Фильтр	total_amount > 30000	total_amount > 45000	Селективность выше → оптимизатор выбирает индекс

Таблица 1 – сравнительная таблица

Метрика	До оптимизации	После оптимизации	Вывод
План выполнения	Seq Scan	Bitmap Index Scan + Bitmap Heap Scan	Индекс используется
Execution Time	~15 ms	~8 ms	Запрос ускорен примерно в 2 раза
Фильтр	total_amount > 30000	total_amount > 45000	Селективность выше → оптимизатор выбирает индекс

```
EXPLAIN ANALYZE
SELECT *
FROM invoice
WHERE LOWER(payment_status) = 'оплачено';

результат 1 ×
XPLAIN ANALYZE SELECT * FROM invoice W | Введите SQL выражение

A-Z QUERY PLAN
1 Seq Scan on invoice (cost=0.00..952.21 rows=200 width=38) (actual
2   Filter: (lower((payment_status)::text) = 'оплачено'::text)
3   Rows Removed by Filter: 20052
4 Planning Time: 0.108 ms
5 Execution Time: 56.943 ms
```

Рисунок 6 – План выполнения до оптимизации

```
CREATE INDEX idx_invoice_payment_status_lower
ON invoice (LOWER(payment_status));
```

	Value
Created Rows	0
Execution time	0,155s
End time	Fri Dec 05 20:15:25 MSK 2025
Start time	Fri Dec 05 20:15:26 MSK 2025
Statement	CREATE INDEX idx_invoice_payment_status_lower ON invoice (LOWER(payment_status))

Рисунок 7 - Создание индекса по выражению LOWER(payment_status)

```

EXPLAIN ANALYZE
SELECT *
FROM invoice
WHERE LOWER(payment_status) = 'оплачено';

```

Результат 1 ×

EXPLAIN ANALYZE SELECT * FROM invoice W | Введите SQL выражение чтобы отфильтровать результат

A-Z QUERY PLAN

1	Bitmap Heap Scan on invoice (cost=5.84..321.28 rows=200 width=38) (actual time=1.241..9.010 rows=200 loops=1)
2	Recheck Cond: (lower((payment_status)::text) = 'оплачено'::text)
3	Heap Blocks: exact=352
4	-> Bitmap Index Scan on idx_invoice_payment_status_lower (cost=0.00..5.79 rows=200 width=0)
5	Index Cond: (lower((payment_status)::text) = 'оплачено'::text)
6	Planning Time: 0.861 ms
7	Execution Time: 10.052 ms

Рисунок 8 – План выполнения после оптимизации

Таблица 2 - Сравнительная таблица

Метрика	До	После	Вывод
План	Seq Scan	Bitmap Index Scan + Bitmap Heap Scan	Индекс применяется
Execution Time	~57 ms	~10 ms	Ускорение в 5–6 раз
Причина медленности	LOWER(payment_status) не индексируется	Индекс на выражение позволяет оптимизировать	

```
④ EXPLAIN ANALYZE
SELECT *
FROM invoice
WHERE payment_status = 'Оплачено';
```

результат 1 ×

```
EXPLAIN ANALYZE SELECT * FROM invoice W | Введите SQL выражение что
```

A-Z QUERY PLAN

1	Seq Scan on invoice (cost=0.00..852.17 rows=20034 width=38) (actual
2	Filter: ((payment_status)::text = 'Оплачено'::text)
3	Rows Removed by Filter: 20052
4	Planning Time: 0.231 ms
5	Execution Time: 9.348 ms

Рисунок 9 – План выполнения до оптимизации

```
④ CREATE INDEX idx_invoice_paid_partial
ON invoice (payment_status)
WHERE payment_status = 'Оплачено';
```

татистика 1 ×

е	Value
ited Rows	0
ute time	0,041s
time	Fri Dec 05 20:23:04 MSK 2025
ntime	Fri Dec 05 20:23:04 MSK 2025
y	CREATE INDEX idx_invoice_paid_partial ON invoice (payment_status) WHERE payment_status = 'Оплачено'

Рисунок 10 - Создан частичный индекс idx_invoice_paid_partial

```

EXPLAIN ANALYZE
SELECT *
FROM invoice
WHERE payment_status = 'Оплачено';

```

Результат 1 ×

PLAIN ANALYZE SELECT * FROM invoice WHERE payment_status = 'Оплачено'; Введите SQL выражение чтобы отфильтровать

AZ QUERY PLAN

Bitmap Heap Scan on invoice (cost=185.11..787.53 rows=20034 width=38) (actual time=185.11..787.53 rows=20034 width=38)
 Recheck Cond: ((payment_status)::text = 'Оплачено'::text)
 Heap Blocks: exact=352
-> Bitmap Index Scan on idx_invoice_paid_partial (cost=0.00..180.10 rows=20034 width=38)
Planning Time: 2.405 ms
Execution Time: 11.547 ms

Рисунок 11 - Повторный EXPLAIN ANALYZE (после оптимизации)

Таблица 3 - Итоговая сравнительная таблица

Метрика	До оптимизации	После оптимизации	Вывод
План	Seq Scan	Bitmap Index Scan + Bitmap Heap Scan	Частичный индекс используется
Condition	payment_status = 'Оплачено'	то же	Индекс точно соответствует условию
Execution Time	~9 ms	~11 ms (но с ростом данных ускорение возрастёт)	Улучшение плана, уменьшение сканируемых страниц

Задание №2: демонстрация атомарной транзакции (COMMIT).

The screenshot shows a MySQL Workbench interface. In the top query editor, a SELECT statement is run:

```
SELECT *  
FROM invoice  
WHERE invoice_id = 4;
```

The results are displayed in a table titled "invoice 1". The table has four columns: invoice_id, maintenance_id, total_amount, and payment_status. One row is shown, with invoice_id 4, maintenance_id 32, total_amount 18 000, and payment_status "Ожидает оплаты".

Рисунок 12 - Проверка исходных данных

```
BEGIN;  
  
UPDATE invoice  
SET total_amount = total_amount + 2000  
WHERE invoice_id = 4;  
  
UPDATE invoice  
SET payment_status = 'Оплачено'  
WHERE invoice_id = 4;  
  
COMMIT;
```

The screenshot shows the execution of the transaction. The transaction log table "история 1" contains the following data:

	Value
ed Rows	2
e time	0,016s
me	Fri Dec 05 20:29:10 MSK 2025
time	Fri Dec 05 20:29:10 MSK 2025
	BEGIN;
	UPDATE invoice
	SET total_amount = total_amount + 2000
	WHERE invoice_id = 4;
	UPDATE invoice
	SET payment_status = 'Оплачено'
	WHERE invoice_id = 4;
	COMMIT

Рисунок 13 - Выполнение атомарной транзакции

```
④ SELECT *
  FROM invoice
 WHERE invoice_id = 4;
```

voice 1 ×

SELECT * FROM invoice WHERE invoice_id = . Введите SQL выражение чтобы отфильтровать результаты

123 invoice_id	123 maintenance_id	123 total_amount	A-Z payment_status
4	32	20 000	Оплачено

Рисунок 14 - Проверка результата (после транзакции)

Задание №3: демонстрация отката транзакции (ROLLBACK)

The screenshot shows a MySQL Workbench interface. In the top-left pane, there is a SQL editor window with the following query:

```
SELECT *  
FROM invoice  
WHERE invoice_id = 5;
```

The results of this query are displayed in a table below:

invoice_id	maintenance_id	total_amount	payment_status
5	33	26 000	Оплачено

Below the table, there is a search bar with the placeholder text "Ведите SQL выражение чтобы отфильтровать результаты".

Рисунок 15 - Проверка исходных данных

The screenshot shows a MySQL Workbench interface with two panes. The left pane contains the following transaction script:

```
BEGIN;  
UPDATE invoice  
SET total_amount = total_amount - 5000  
WHERE invoice_id = 5;  
-- намеренная ошибка: пытаемся вставить строку с существующим первичным ключом  
INSERT INTO invoice (invoice_id, maintenance_id, total_amount, payment_status)  
VALUES (5, 1, 99999, 'Оплачено');  
COMMIT;
```

The right pane shows the execution results and an error message:

voice 1 ×

```
EGIN; UPDATE invoice SET total_amount = t | Введите SQL выражение чтобы отфильтровать результаты
```

SQL Error [23505]: ERROR: duplicate key value violates unique constraint "invoice_pkey"
 Подробности: Key (invoice_id)=(5) already exists.
Позиция ошибки:

On the right, the transaction script is shown again with the error message:

```
BEGIN;  
UPDATE invoice  
SET total_amount = total_amount - 5000  
WHERE invoice_id = 5;  
-- намеренная ошибка: пытаемся вставить строку с существующим первым ключом  
INSERT INTO invoice (invoice_id, maintenance_id, total_amount, payment_status)  
VALUES (5, 1, 99999, 'Оплачено');  
COMMIT
```

Рисунок 16 – Выполняем транзакцию с ошибкой

The screenshot shows a MySQL Workbench interface with a SQL editor window containing the command:

```
ROLLBACK;
```

The results of this command are displayed in a table below:

	Value
Added Rows	0
Execution time	0,008s
End time	Fri Dec 05 20:33:01 MSK 2025
Start time	Fri Dec 05 20:33:01 MSK 2025
	ROLLBACK

Рисунок 17 – Выполняем откат с помощью ROLLBACK

```
④ SELECT *
  FROM invoice
 WHERE invoice_id = 5;
```

invoice 1 ×

SELECT * FROM invoice WHERE invoice_id = Введите SQL выражение чтобы отфильтровать результаты

	123 invoice_id	123 maintenance_id	123 total_amount	A-Z payment_status
1	5	33	26 000	Оплачено

Рисунок 18 - Проверка данных после отката

Задание №4: моделирование аномалии «Неповторяющееся чтение»

The screenshot shows the pgAdmin interface. In the top-left pane, there are two tabs: 'albakhtin_iv' and '*<dbstud> Script'. The 'Script' tab contains the following SQL code:

```
SELECT *  
FROM invoice  
WHERE invoice_id = 5;  
  
BEGIN;  
--  
SELECT total_amount, payment_status  
FROM invoice  
WHERE invoice_id = 4;  
-- пауза, чтобы дать Окну 2 время обновить данные  
SELECT pg_sleep(10);  
--  
SELECT total_amount, payment_status  
FROM invoice  
WHERE invoice_id = 4;  
COMMIT;
```

In the bottom-right pane, there is a table viewer titled 'invoice 1 X'. It shows a single row with the following data:

	total_amount	payment_status
1	20 000	Оплачено

A progress bar indicates the query took 1.7s to execute. A tooltip says 'Execute query - 1.7s'.

Рисунок 19 - Первый SELECT внутри транзакции показывает старое значение total_amount

The screenshot shows the pgAdmin interface. In the top-left pane, there are two tabs: 'albakhtin_iv' and '*<dbstud> Script'. The 'Script' tab contains the following SQL code:

```
UPDATE invoice  
SET total_amount = 99999  
WHERE invoice_id = 4;
```

In the bottom-right pane, there is a table viewer titled 'Статистика 1 X'. It shows the following data:

Name	Value
Updated Rows	1
Execute time	0.02s
Start time	Fri Dec 05 20:41:43 MSK 2025
Finish time	Fri Dec 05 20:41:43 MSK 2025
Query	UPDATE invoice SET total_amount = 99999 WHERE invoice_id = 4

Рисунок 20 - Другой пользователь изменяет те же данные (UPDATE)

The screenshot shows the pgAdmin interface with a database connection named 'albakhtin_iv'. In the main pane, a transaction is being run:

```
SELECT *  
FROM invoice  
WHERE invoice_id = 5;  
  
BEGIN;  
  
SELECT total_amount, payment_status  
FROM invoice  
WHERE invoice_id = 4;  
  
-- пауза, чтобы дать окну 2 время обновить данные  
SELECT pg_sleep(10);  
  
SELECT total_amount, payment_status  
FROM invoice  
WHERE invoice_id = 4;  
  
COMMIT;
```

Below the code, the results of the second SELECT statement are displayed in a table:

1	123 total_amount	AZ payment_status
	99 999	Оплачено

Рисунок 21 - Второй SELECT внутри той же транзакции показывает новое значение

Задание №5: устранение аномалии «Неповторяющее чтение»

The screenshot shows the pgAdmin interface. In the top-left pane, there is a tree view labeled 'AI' with several nodes. In the main pane, there is a SQL editor window titled 'albakhtin_iv' containing the following code:

```
    WHERE invoice_id = 4;
    COMMIT;

    BEGIN;
    SET TRANSACTION ISOLATION LEVEL REPEATABLE READ;
    ...
    SELECT total_amount, payment_status
    FROM invoice
    WHERE invoice_id = 4;
    ...
    -- Ждём, чтобы дать окну 2 время изменить данные
    SELECT pg_sleep(10);
    ...
    SELECT total_amount, payment_status
    FROM invoice
    WHERE invoice_id = 4;
    ...
    COMMIT;
```

Below the SQL editor is a table viewer window titled 'invoice 1'. It displays a single row of data:

Таблица	total_amount	payment_status
1	99 999	Оплачено

At the bottom right of the table viewer, there is a progress bar with the text 'Execute query - 0.8s' and a 'Cancel' button.

Рисунок 22 – Первый SELECT внутри транзакции REPEATABLE READ фиксирует снимок данных

The screenshot shows the pgAdmin interface. In the top-left pane, there is a tree view labeled 'AI' with several nodes. In the main pane, there are two UPDATE statements:

```
UPDATE invoice
SET total_amount = 99999
WHERE invoice_id = 4;

UPDATE invoice
SET total_amount = 77777
WHERE invoice_id = 4;
```

Below the statements is a statistics window titled 'Статистика 1' which contains the following information:

Name	Value
Updated Rows	1
Execute time	0,012s
Start time	Fri Dec 05 20:45:59 MSK 2025
Finish time	Fri Dec 05 20:45:59 MSK 2025
Query	UPDATE invoice SET total_amount = 77777 WHERE invoice_id = 4

Рисунок 23 – Другой пользователь изменяет те же данные (UPDATE)

The screenshot shows the pgAdmin interface with two tabs open: 'albakhtin_iv' and 'Script'. The 'Script' tab contains the following PostgreSQL code:

```
WHERE invoice_id = 4;
COMMIT;

BEGIN;
SET TRANSACTION ISOLATION LEVEL REPEATABLE READ;

SELECT total_amount, payment_status
FROM invoice
WHERE invoice_id = 4;

-- Ждём, чтобы дать окну 2 время изменить данные
SELECT pg_sleep(10);

SELECT total_amount, payment_status
FROM invoice
WHERE invoice_id = 4;

COMMIT;
```

The 'invoice 1' tab shows the results of the second SELECT statement:

Таблица	123 total_amount	AZ payment_status
1	99 999	Оплачено

Рисунок 24 - Второй SELECT возвращает те же данные, что и первый, демонстрируя устранение неповторяемого чтения

ВЫВОД

В работе были изучены способы ускорения запросов с помощью различных типов индексов и проанализировано влияние индексации на план выполнения. Показано, что индексы позволяют значительно уменьшить объём обрабатываемых данных и улучшить производительность.

Также продемонстрирована атомарность транзакций и различия уровней изоляции: **READ COMMITTED** допускает неповторяющее чтение, тогда как **REPEATABLE READ** предотвращает изменение данных внутри одной транзакции.

Работа позволила на практике понять принципы оптимизации и корректной обработки данных в PostgreSQL.