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 «Неповторяемое чтение».

Используя два редактора SQL, смоделировать проблему «неповторяемое 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 «Неповторяемое 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.

ХОД ВЫПОЛНЕНИЯ РАБОТЫ

Введение

Для создания понятных и последовательных примеров в рамках данной работы будет использоваться упрощенная схема базы данных «Аптека».

Эта схема послужит основой для демонстрации работы индексов и транзакций. Студентам следует адаптировать представленные примеры для своих собственных баз данных.

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
5	5	Активированный уголь	500	25	2025-07-10	2030-07-10	1
6	8	Корвалол	140	45	2025-06-01	2029-06-01	1

Таблица 3. Таблица sale_items (Проданные товары)

	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 customer_audit (Аудит)

	user_id	first_name	last_name	phone_number	change_date
1	1	Иван	Иванов	+79001234567	2025-10-20 01:26:02.183

Таблица 5. Таблица customers (Клиенты)

	customer_id	first_name	last_name	email	phone_number
1	2	Петр	Петров	petr@example.com	[NULL]
2	1	Иван	Иванов	ivan@example.com	+79998887766

ПОДГОТОВКА БАЗЫ ДАННЫХ

Анализ производительности **EXPLAIN** имеет смысл только на **большом объеме данных**. Если в вашей таблице 10-20 строк, PostgreSQL всегда выберет Seq Scan (полное сканирование), так как это быстрее, чем даже загружать индекс.

Чтобы увидеть реальный эффект от оптимизации, в ключевой таблице должны быть тысячи строк.

Алгоритм генерации данных для вашей БД

1. **Определите таблицу.** Выберите главную «сущностную» таблицу в вашей БД (например, products, articles, employees).
2. **Используйте анонимный блок.** Мы будем использовать конструкцию `DO $$ ... END;` (анонимный блок **PL/pgSQL**) для выполнения цикла.
3. **Используйте цикл FOR LOOP** для генерации 20 000 строк.
4. **Используйте INSERT**. Внутри цикла будет выполняться INSERT с конкатенацией (соединением) строки и номера итерации: `'Название №'||i`
5. **Используйте random()** для генерации случайных числовых данных (цены, количества).

Ниже приведен пример такого скрипта для демонстрационной БД «Аптека». Вы должны адаптировать его для своей структуры таблиц.

Листинг 1. Пример скрипта для генерации 20 000+ строк в medicines

```
DO $$
BEGIN
  FOR i IN 1..20000 LOOP
    INSERT INTO medicines (name, quantity_in_stock, price,
manufacturer_id)
    VALUES (
      'Лекарство №' || i,           -- Название с номером лекарства
      (random() * 50)::INT,         -- Количество от 0 до 50
      (random() * 100 + 10)::DECIMAL(10,2), -- Цена от 10 до 110
      (random() * 1 + 1)::INT       -- ID производителя (1 или 2)
    );
  END LOOP;
END;
$$ LANGUAGE plpgsql;
```

ВАЖНО: учтите, что для корректной работы скрипта необходимо, чтобы используемым в качестве внешнего ключа ID производителя существовал. В скрипте выше именно поэтому допустимое значение ограничено 1 или 2.

После выполнения этого скрипта продемонстрируйте результат с помощью `SELECT COUNT(*) FROM medicines;` – он должен показать 20 000+ строк.

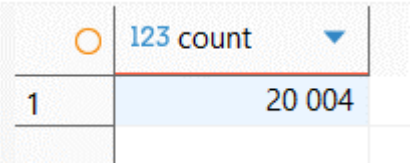


Рисунок 0 – Количество записей в таблице

Теперь можно проводить анализ.

1. АНАЛИЗ И ОПТИМИЗАЦИЯ SQL-ЗАПРОСОВ

1.1 Анализ «медленных» запросов (EXPLAIN ANALYZE)

Когда вы отправляете запрос, оптимизатор СУБД анализирует его и решает, как его лучше выполнить, составляя ***план выполнения*** (**ПВ**).

Команда **EXPLAIN ANALYZE** не просто показывает предполагаемый план, а **реально выполняет запрос**, замеряя **фактическое время**.

При **анализе плана** нас в первую очередь **интересует:**

- **Seq Scan (Sequential Scan)** – это **«плохой»** оператор. Он означает, что PostgreSQL для поиска данных **последовательно прочитал всю таблицу** от начала до конца.
- **Index Scan** – это **«хороший»** оператор. Он означает, что СУБД **использовала индекс** для **мгновенного поиска** нужных строк.
- **Execution Time** – **фактическое время выполнения** запроса в миллисекундах.

1.2 Сценарий 1. B-Tree (стандартный поиск)

Индекс – это специальная структура данных, которая позволяет СУБД быстро находить строки, минуя полное сканирование таблицы.

B-Tree (B-дерево) – это тип индекса, используемый в PostgreSQL **по умолчанию**.

Его можно сравнить с **предметным указателем** в конце книги: вместо того чтобы листать **всю книгу** (**Seq Scan**), вы **смотрите в указатель** (**Index Scan**) и сразу переходите на нужную страницу.

Он идеально подходит для операций равенства (=), сравнения (>) и диапазонов (**BETWEEN**).

Задача:

Найти в таблице medicines (10 000+ строк) лекарство по его точному названию.

Шаг 1. Анализ «ДО» *(без индекса)*

Выполняем **EXPLAIN ANALYZE** для запроса.

Листинг 2. Анализируем запрос

```
EXPLAIN ANALYZE SELECT * FROM medicines WHERE name = 'Аспирин';
```

Оцениваем результаты.

	QUERY PLAN
1	Seq Scan on medicines (cost=0.00..219.72 rows=13 width=544) (actual time=0.015..2.361 rows=1 loops=1
2	Filter: ((name)::text = 'Аспирин'::text)
3	Rows Removed by Filter: 20003
4	Planning Time: 0.077 ms
5	Execution Time: 2.374 ms

Рисунок 1 – Результат запроса

Проводим анализ:

Тип плана – Seq Scan.

PostgreSQL прочитал **ВСЕ** 20 000+ строк (Rows Removed by Filter: 20003).

Время выполнения – Execution Time: **2.374 ms**.

Это время будет **линейно расти** с увеличением таблицы.

Шаг 2. Создание индекса

Создаём стандартный *(создаётся по умолчанию)* **B-Tree** индекс.

Листинг 3. Создаём B-Tree индекс

```
CREATE INDEX idx_medicines_name ON medicines(name);
```

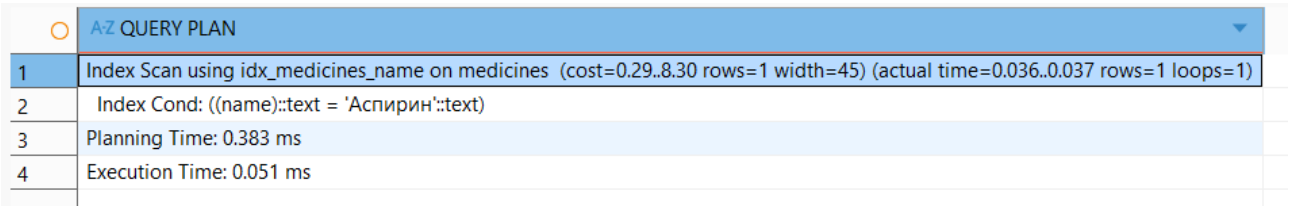
Шаг 3. Анализ «ПОСЛЕ» (с индексом)

Выполняем *тот же самый* запрос **EXPLAIN ANALYZE** ещё раз.

Листинг 4. Повторно анализируем запрос

```
EXPLAIN ANALYZE SELECT * FROM medicines WHERE name = 'Аспирин';
```

Оцениваем результаты.



	QUERY PLAN
1	Index Scan using idx_medicines_name on medicines (cost=0.29..8.30 rows=1 width=45) (actual time=0.036..0.037 rows=1 loops=1)
2	Index Cond: ((name)::text = 'Аспирин'::text)
3	Planning Time: 0.383 ms
4	Execution Time: 0.051 ms

Рисунок 2 – Результат запроса

Проводим анализ:

Тип плана – Index Scan.

PostgreSQL использовал наш индекс.

Обратите внимание: **отфильтрованные** строки **пропали**.

Время выполнения – Execution Time: **0.051 ms**.

Это время будет **незначительно расти** с увеличением таблицы.

Шаг 4. Сравнительная таблица

Составим сравнительную таблицу для наглядной демонстрации времени выполнения запросов.

Метрика	**До оптимизации**	**После оптимизации**	**Вывод**
План (Оператор)	Seq Scan	Index Scan	Выбор отличается
Execution Time	**2.374 ms**	**0.051 ms**	Запрос **ускорился** примерно **в 46 раз**.

Важно помнить, что чем больше данных будет в таблице, тем значительнее будет ускорение.

1.3 Сценарий 2. Индекс по выражению *(Function-based)*

Обычный индекс **B-Tree** (как idx_medicines_name) **не будет использоваться**, если вы применяете к столбцу какую-либо функцию в **WHERE,** например **LOWER()**.

Для решения таких задач, используется индекс по выражению *(или «индекс на вычисляемых столбцах»)*.

Он индексирует **не сам столбец** (name), а **результат функции** (в нашем случае – ***LOWER(name)***).

Без этой функции поиск товара ***«аспирин»*** вернул бы нам 0 совпадений… Ведь товар в базе сохранён как ***«Аспирин»***, а так как коды символов «***а***» и «***А***» – разные, то и поиск найдёт только точное совпадение.

Задача:

Найти нужное лекарство, в каком бы регистре оно ни было сохранено в таблице.

Шаг 1. Анализ «ДО» (без индекса)

Выполняем **EXPLAIN ANALYZE** для запроса.

Листинг 5. Анализируем запрос

```
EXPLAIN ANALYZE SELECT * FROM medicines WHERE LOWER(name) = 'аспирин';
```

Оцениваем результаты.

	QUERY PLAN
1	Seq Scan on medicines (cost=0.00..487.06 rows=100 width=45) (actual time=0.028..14.576 rows=1 loops=1)
2	Filter: (lower((name)::text) = 'аспирин'::text)
3	Rows Removed by Filter: 20003
4	Planning Time: 0.082 ms
5	Execution Time: 14.593 ms

Рисунок 3 – Результат запроса

Проводим анализ:

Тип плана – Seq Scan.

PostgreSQL прочитал **ВСЕ** 20 000+ строк (Rows Removed by Filter: 20003).

Время выполнения – Execution Time: **14.593 ms**.

Это время будет **линейно расти** с увеличением таблицы.

Шаг 2. Создание индекса

Создаём индекс.

Листинг 6. Создаём B-Tree индекс

```
CREATE INDEX idx_medicines_name_lower ON medicines (LOWER(name));
```

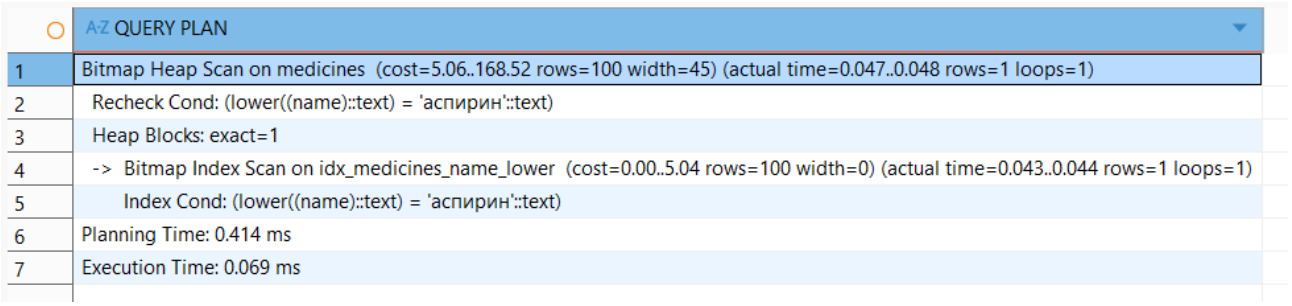
Шаг 3. Анализ «ПОСЛЕ» (с индексом)

Выполняем *тот же самый* запрос **EXPLAIN ANALYZE** ещё раз.

Листинг 7. Повторно анализируем запрос

```
EXPLAIN ANALYZE SELECT * FROM medicines WHERE LOWER(name) = 'аспирин';
```

Оцениваем результаты.



	QUERY PLAN
1	Bitmap Heap Scan on medicines (cost=5.06..168.52 rows=100 width=45) (actual time=0.047..0.048 rows=1 loops=1)
2	Recheck Cond: (lower((name)::text) = 'аспирин'::text)
3	Heap Blocks: exact=1
4	-> Bitmap Index Scan on idx_medicines_name_lower (cost=0.00..5.04 rows=100 width=0) (actual time=0.043..0.044 rows=1 loops=1)
5	Index Cond: (lower((name)::text) = 'аспирин'::text)
6	Planning Time: 0.414 ms
7	Execution Time: 0.069 ms

Рисунок 4 – Результат запроса

Проводим анализ:

Тип плана – Bitmap Index Scan.

PostgreSQL использовал наш индекс.

Время выполнения – Execution Time: **0.069 ms**.

Это время будет **незначительно расти** с увеличением таблицы.

Шаг 4. Сравнительная таблица

Составим сравнительную таблицу для наглядной демонстрации времени выполнения запросов.

Метрика	**До оптимизации**	**После оптимизации**	**Вывод**
План (Оператор)	Seq Scan	Bitmap Index Scan	Выбор отличается
Execution Time	**14.593 ms**	**0.069 ms**	Запрос **ускорился** примерно **в 211 раз**.

Здесь ускорение ещё значительнее, а ведь это – всего 20 тысяч строк. В реальных таблицах их могут быть миллионы, и даже больше.

1.4 Сценарий 3. Частичный *(отфильтрованный)* индекс

Иногда вы ищете не любые данные, а маленькое, специфическое подмножество *(например,* ***is_archived = true****, или* ***quantity = 0****)*.

Обычный B-Tree индекс окажется «раздут», храня огромное множество ненужных нам данных.

В таких случаях применяется **частичный *(отфильтрованный)* индекс**.

Он индексирует только те строки, которые соответствуют заданному в **WHERE** условию, что делает такой индекс очень маленьким и быстрым.

Задача:

Найти все лекарства, которых нет на складе (quantity_in_stock = 0).

Шаг 1. Анализ «ДО» (без индекса)

Выполняем **EXPLAIN ANALYZE** для запроса.

Листинг 8. Анализируем запрос

```
EXPLAIN ANALYZE SELECT * FROM medicines WHERE quantity_in_stock = 0;
```

Оцениваем результаты.

	QUERY PLAN
1	Seq Scan on medicines (cost=0.00..437.05 rows=192 width=45) (actual time=0.018..2.204 rows=192 loops=1)
2	Filter: (quantity_in_stock = 0)
3	Rows Removed by Filter: 19812
4	Planning Time: 0.058 ms
5	Execution Time: 2.217 ms

Рисунок 5 – Результат запроса

Проводим анализ:

Тип плана – Seq Scan.

PostgreSQL прочитал **ВСЕ** 20 000+ строк (Rows Removed by Filter: 19812), чтобы найти всего **192 подходящих**.

Время выполнения – Execution Time: **2.217 ms**.

Это время будет **линейно расти** с увеличением таблицы.

Шаг 2. Создание индекса

Создаём стандартный **B-Tree** индекс.

Листинг 9. Создаём B-Tree индекс

```
CREATE INDEX idx_medicines_out_of_stock ON medicines (id)
WHERE quantity_in_stock = 0;
```

Шаг 3. Анализ «ПОСЛЕ» (с индексом)

Выполняем *тот же самый* запрос **EXPLAIN ANALYZE** ещё раз.

Листинг 10. Повторно анализируем запрос

```
EXPLAIN ANALYZE SELECT * FROM medicines WHERE quantity_in_stock = 0;
```

Оцениваем результаты.

	QUERY PLAN
1	Index Scan using idx_medicines_out_of_stock on medicines (cost=0.14..16.03 rows=192 width=45) (actual time=0.032..0.149 rows=192 loops=1)
2	Planning Time: 0.526 ms
3	Execution Time: 0.176 ms

Рисунок 6 – Результат запроса

Проводим анализ:

Тип плана – Index Scan.

PostgreSQL использовал наш индекс.

Время выполнения – Execution Time: **0.176 ms**.

Это время будет **незначительно расти** с увеличением таблицы.

Шаг 4. Сравнительная таблица

Составим сравнительную таблицу для наглядной демонстрации времени выполнения запросов.

Метрика	**До оптимизации**	**После оптимизации**	**Вывод**
План (Оператор)	Seq Scan	Index Scan	Выбор отличается
Execution Time	**2.217 ms**	**0.176 ms**	Запрос **ускорился** примерно **в 12.5 раз**.

Здесь разница кажется не такой большой, однако стоит помнить, что здесь всего 20 тысяч строчек, а чем их будет больше, тем значительнее будет становиться разница.

2. УПРАВЛЕНИЕ ТРАНЗАКЦИЯМИ

2.1 Основы транзакций (ACID)

Транзакция — это логическая, неделимая последовательность операций, которая переводит базу данных из одного целостного состояния в другое.

Представьте **банковский перевод**:

- `UPDATE users SET balance = balance - 100 WHERE name = 'A';`
- `UPDATE users SET balance = balance + 100 WHERE name = 'B';`

Обе операции должны либо **успешно выполниться вместе**, либо **провалиться вместе**. Нельзя допустить, чтобы деньги **снялись со счета А**, но **не дошли до счета Б**. Транзакция гарантирует это.

Работа транзакций основана на принципах ACID:

- **A (Atomicity / Атомарность)** – «Всё или ничего».
- **C (Consistency / Согласованность)** – БД всегда остается в корректном состоянии.
- **I (Isolation / Изолированность)** – параллельные транзакции не мешают друг другу.
- **D (Durability / Долговечность)** – если транзакция завершена, ее изменения постоянны.

Для управления транзакциями используются три команды:

- **BEGIN** – начать транзакцию.
- **COMMIT** – успешно завершить и сохранить все изменения.
- **ROLLBACK** – отменить все изменения, сделанные с момента BEGIN.

2.2 Демонстрация атомарности. Успешный COMMIT

Задача:

Продать 10 «Аспирина» (ID=2) для чека (ID=1).

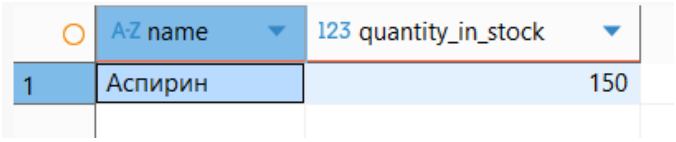
Шаг 1. Анализ «ДО»

Для начала, проверим исходные данные.

Листинг 11. Проверка исходных данных

```
SELECT name, quantity_in_stock FROM medicines WHERE id = 2;
SELECT * FROM sale_items;
```

Как видим, у нас в наличии есть 150 Аспирина, а в таблице ***sale_items*** есть 2 записи.



	A-Z name	123 quantity_in_stock
1	Аспирин	150

Рисунок 7 – Количество «Аспирина»

	123 sale_item_id	123 sale_id	123 medicine_id	123 quantity	123 unit_price
1	1	1	1	3	50,5
2	2	2	2	1	120

Рисунок 8 – Количество записей в sale_items

Шаг 2. Выполняем атомарную операцию

Выполним списание товара и добавим запись в чек.

Здесь мы выполняем сразу две операции, и чтобы показать, что они или должны быть выполнены обе, или никто из них – помещаем их в транзакцию. Для этого мы помещаем их код между операторами **BEGIN;** и **COMMIT;**.

Листинг 12. Выполняем атомарную операцию

```
BEGIN;
     UPDATE medicines
     SET quantity_in_stock = quantity_in_stock - 10
     WHERE id = 2;

     INSERT INTO sale_items (sale_id, medicine_id, quantity,
unit_price)
     VALUES (1, 2, 10, 50.50);
COMMIT;
```

Шаг 3. Анализ «ПОСЛЕ»

Теперь проверим, каков результат. Если оба оператора выполнились без ошибок, **COMMIT** фиксирует изменения окончательно, и мы их увидим.

Листинг 13. Повторно проверим данные

```
SELECT name, quantity_in_stock FROM medicines WHERE id = 2;
SELECT * FROM sale_items;
```

Аспирина осталось 140 единиц, что логично.

	A-Z name	123 quantity_in_stock
1	Аспирин	140

Рисунок 9 – Результат запроса

В ***sale_items*** появилась новая запись – тоже всё хорошо.

	123 sale_item_id	123 sale_id	123 medicine_id	123 quantity	123 unit_price
1	1	1	1	3	50,5
2	2	2	2	1	120
3	3	1	2	10	50,5

2.3 Демонстрация атомарности. Аварийный ROLLBACK

Задача:

Показать, что при ошибке внутри транзакции она целиком считается неуспешной и должна быть полностью отменена с помощью **ROLLBACK**, а сервер до явного решения владельца соединения не завершает её автоматически.

Шаг 1. Выполним атомарную операцию с ошибкой

Мы повторим выполнение шага 2, однако во втором запросе допустим намеренную ошибку – укажем ID записи, которую нужно добавить в таблицу ***sale_items***, и этот ID **уже** будет существовать.

В реальности такие ошибки, безусловно, вряд ли будут допущены, однако это продемонстрирует что будет, если первая операция выполнилась успешно, но при выполнении второй операции произошла ошибка.

Листинг 14. Выполним запрос с ошибкой

```
BEGIN;
     UPDATE medicines
     SET quantity_in_stock = quantity_in_stock - 10
     WHERE id = 2;

     INSERT INTO sale_items (sale_item_id, sale_id, medicine_id,
quantity, unit_price)
     VALUES (1, 1, 2, 200, 50.50);
COMMIT;
```

При выполнении мы сразу же получаем ошибку, что вполне логично.

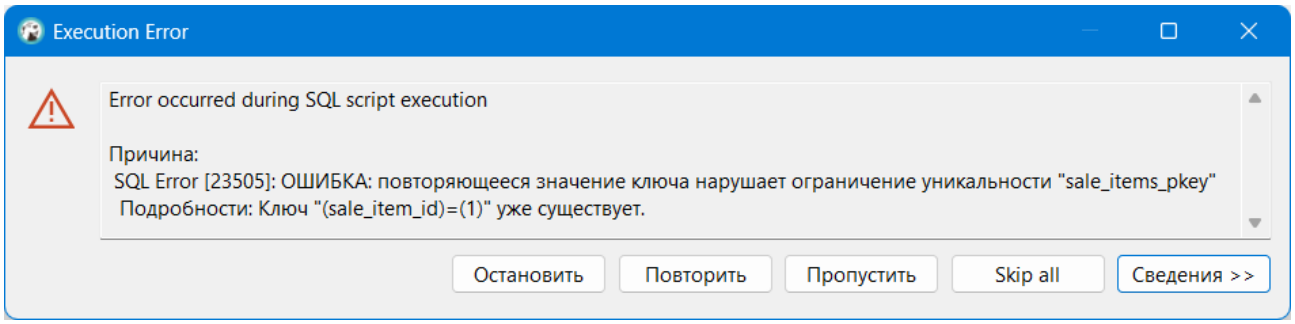


Рисунок 10 – Ошибка при попытке добавить запись с уже существующим sale_item_id.

Происходит следующее:

- **Команда UPDATE** отрабатывает успешно, но её изменения пока только внутри транзакции (для других пользователей они не видны).
- **Команда INSERT** вызывает ошибку: нарушено ограничение уникальности/первичного ключа.
- После этой ошибки **вся транзакция** помечается PostgreSQL как **«прерванная» (aborted)**.
- Команда **COMMIT** уже **не будет выполнена**. Соединение остаётся «внутри» ошибочной транзакции.

Важно понимать: PostgreSQL не выполняет за нас автоматический **ROLLBACK** в явном блоке **BEGIN**…**COMMIT**. Он **переводит транзакцию в состояние ошибки *(aborted)*** и дальнейшие операции невозможны без явного вызова **ROLLBACK**.

Шаг 2. Анализ «ПОСЛЕ»

Попробуем, не выполняя ROLLBACK, запросить данные.

Листинг 15. Запросим актуальные данные

```
SELECT name, quantity_in_stock FROM medicines WHERE id = 2;
SELECT * FROM sale_items;
```

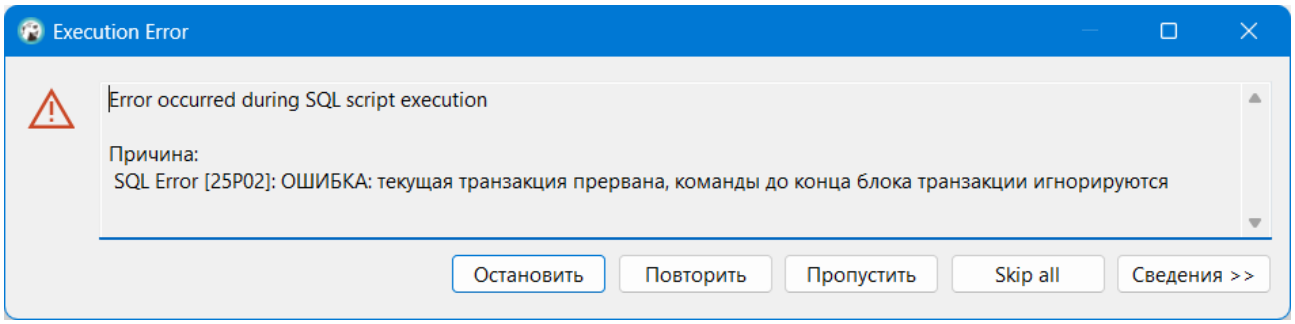


Рисунок 11 – Результат запроса

Как мы видим, предыдущий запрос ещё не завершился – транзакция находится в состоянии **ошибки** (**aborted**). Перед выполнением следующего запроса, её нужно вывести из этого состояния.

В состоянии ошибки:

- Любой следующий запрос в этом соединении вернёт ошибку вроде той, что мы получили выше.
- Реальных изменений в базе данных ещё нет: ни уменьшения остатка, ни новой строки в sale_items другие транзакции не увидят.
- Соединение как бы «зависло» в прерванной транзакции и ждёт, что мы либо её откатим, либо закроем соединение.

Чтобы выйти из этого состояния, нужно завершить транзакцию явным откатом. Выполним соответствующий запрос в редакторе DBeaver.

Листинг 16. Явно завершаем транзакцию

```
ROLLBACK;
```

Эту команду мы выполняем отдельным запросом (например, в DBeaver — в той же вкладке, где у нас выполнялся BEGIN/COMMIT).

После этого соединение «очищено», и мы можем снова выполнять любые запросы.

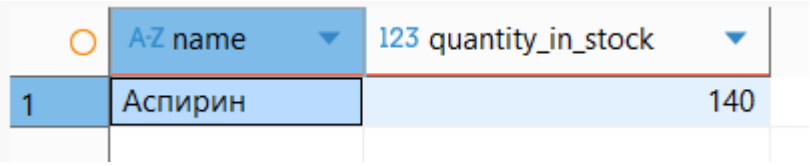
Шаг 3. Проверка состояния после ROLLBACK

Снова запросим актуальные данные.

Листинг 17. Повторно запросим актуальные данные

```
SELECT name, quantity_in_stock FROM medicines WHERE id = 2;
SELECT * FROM sale_items;
```

Аспирина осталось всё также осталось 140 единиц, транзакция отменила все ранее совершённые операции, которые были указаны после `BEGIN;`.



	A-Z name	123 quantity_in_stock
1	Аспирин	140

Рисунок 12 – Результат запроса

В ***sale_items*** ничего не изменилось – здесь тоже всё в порядке.

	123 sale_item_id	123 sale_id	123 medicine_id	123 quantity	123 unit_price
1	1	1	1	3	50,5
2	2	2	2	1	120
3	3	1	2	10	50,5

То есть все изменения, сделанные **после** **BEGIN**, были **полностью отменены**.

Это и есть проявление атомарности: несмотря на то, что **UPDATE** формально выполнился раньше, он **не был зафиксирован** и ушёл в откат вместе с неудачным **INSERT**.

Использование в реальных проектах

В реальных приложениях **ROLLBACK** чаще всего вызывается не «руками» в DBeaver, а серверным кодом (драйвер/ORM), когда операция завершается с ошибкой.

Важно понимать, что PostgreSQL не будет сам откатывать **явную** транзакцию *(т.е. ту, которую мы запускаем с помощью BEGIN)*. Он будет ждать решения владельца соединения: выполнить **ROLLBACK** или **COMMIT**.

Листинг 18. Пример вызова ROLLBACK в реальных проектах (пример на Python)

```
import psycopg2

def do_transaction():
  with psycopg2.connect(
    host="localhost",
    database="pharmacy",
    user="postgres",
    password="password"
  ) as conn:

      with conn.cursor() as cur:
        try:
          cur.execute("UPDATE medicines SET
quantity_in_stock = quantity_in_stock - 10 WHERE id = 2;")
          cur.execute("INSERT INTO sale_items VALUES (1, 1,
2, 10, 50.50);")
# commit произойдёт автоматически при выходе из `with conn:`
        except Exception as e:
          print("Ошибка:", e)
          conn.rollback()   # обязательный manual rollback
          raise
```

2.4 Моделирование аномалии «Неповторяемое чтение»

При параллельной работе нескольких транзакций могут возникать аномалии.

«Неповторяемое чтение» (Non-Repeatable Read) — это аномалия, при которой транзакция читает одни и те же данные дважды и получает разные результаты. Это происходит потому, что в промежутке между чтениями другая транзакция успела изменить эти данные и зафиксировать их (COMMIT).

Уровень изоляции PostgreSQL по умолчанию, **READ COMMITTED** («чтение зафиксированного»), допускает эту аномалию.

Задача:

Показать, что **READ COMMITTED** уязвим, используя два отдельных скрипта.

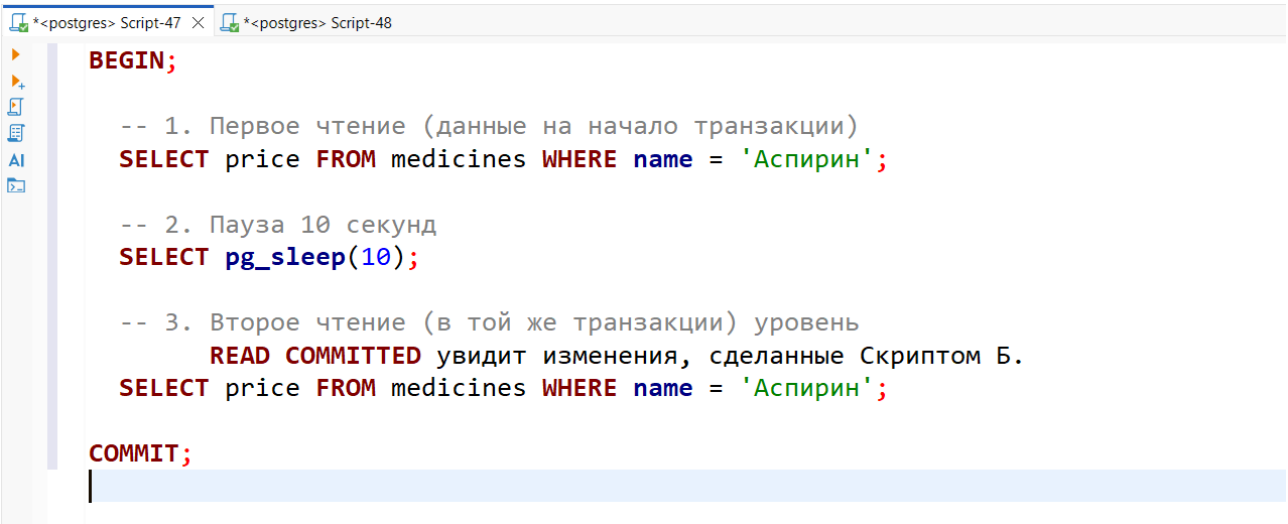
Решение:

Мы используем функцию ***pg_sleep(10)***, чтобы искусственно замедлить Скрипт А. Это создаст нам 10-секундное «окно» для запуска Скрипта Б.

Инструкция:

1. **Запустите Скрипт А в первом окне**. Он «зависнет» на 10 секунд.
2. Пока **Скрипт А** выполняется, быстро **перейдите во второе окно** и **запустите Скрипт Б**.
3. **Вернитесь** в окно **Скрипта А**, и **после завершения** изучите **все вкладки результата**.

Откройте две вкладки редактора, и добавьте в первую Скрипт А, во вторую – Скрипт Б *(на скриншоте ниже – вкладки Script-47 и Script-48).*



Адаптируйте скрипты под вашу базу данных, и добавить в отдельные вкладки.

Листинг 19. Скрипт А, для выполнения в первом окне.

```
BEGIN;

  -- 1. Первое чтение (данные на начало транзакции)
  SELECT price FROM medicines WHERE name = 'Аспирин';

  -- 2. Пауза 10 секунд
  SELECT pg_sleep(10);
```

```
  -- 3. Второе чтение (в той же транзакции) уровень
  --    READ COMMITTED увидит изменения, сделанные Скриптом Б.
  SELECT price FROM medicines WHERE name = 'Аспирин';

COMMIT;
```

Листинг 20. Скрипт Б, для выполнения во втором окне.

```
-- Повышаем цену
UPDATE medicines SET price = 12000.00 WHERE name = 'Аспирин';
```

Посмотрим на результаты. Нас интересуют вкладки 1 и 3.

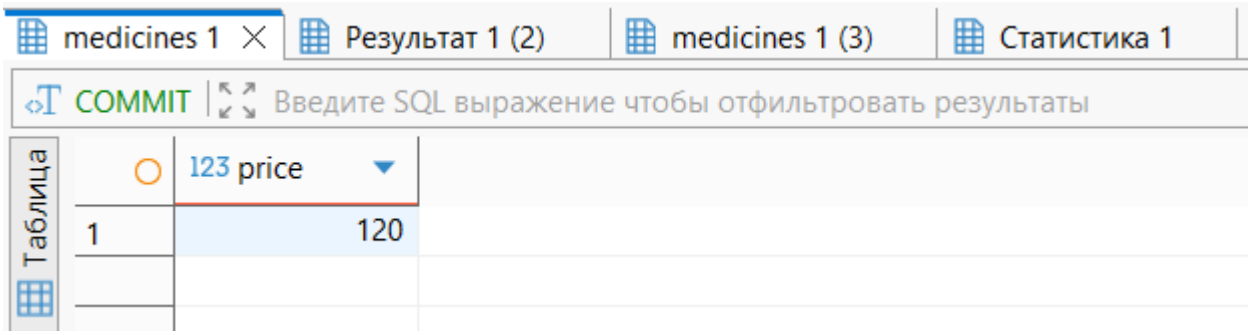


Рисунок 13 – Количество «Аспирина» при первом чтении

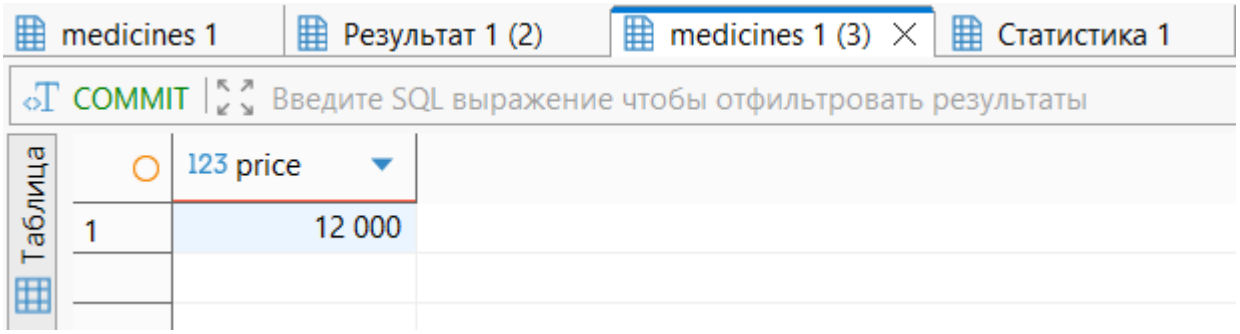


Рисунок 14 – Количество «Аспирина» при повторном чтении, спустя 10 секунд

Вывод:

Как видим, первое чтение показало значение 120, а второе чтение *(в той же транзакции)* показало 12000.

Данные «не повторились», и при работе над реальным проектом, это может привести к серьёзным проблемам.

Аномалия успешно смоделирована.

2.5 Устранение аномалии «Неповторяемое чтение»

Для борьбы с этой аномалией используется более строгий уровень изоляции — **REPEATABLE READ** *(«повторяемое чтение»)*.

Как он работает:

На этом уровне транзакция работает с «**моментальным снимком**» ***(snapshot)*** данных, сделанным **в момент начала транзакции**.

Она не видит любые изменения, зафиксированные другими транзакциями *(в том числе одиночными запросами)* после этого снимка.

Задача:

Доказать, что **REPEATABLE READ** решает эту проблему.

Решение:

Перед выполнением не забудьте «откатить» цену на ту, которая была ранее *(или любую другую)*.

Листинг 21. Откатываем цену.

```
UPDATE medicines SET price = 120.00 WHERE name = 'Аспирин';
```

Добавляем строчку с указанием уровня изоляции **REPEATABLE READ** в Скрипт А.

Листинг 22. Скрипт А, для выполнения в первом окне (обновлённая версия).

```
BEGIN;

  SET TRANSACTION ISOLATION LEVEL REPEATABLE READ;

  -- 1. Первое чтение (данные на начало транзакции)
  SELECT price FROM medicines WHERE name = 'Аспирин';

  -- 2. Пауза 10 секунд
  SELECT pg_sleep(10);

  -- 3. Второе чтение (в той же транзакции) уровень
  --    REPEATABLE READ НЕ увидит изменений, сделанных Скриптом Б
  SELECT price FROM medicines WHERE name = 'Аспирин';

COMMIT;
```

Листинг 23. Скрипт Б, для выполнения во втором окне (БЕЗ ИЗМЕНЕНИЙ).

```
-- Повышаем цену
```

```
UPDATE medicines SET price = 12000.00 WHERE name = 'Аспирин';
```

Посмотрим на результаты. Нас всё также интересуют вкладки 1 и 3.

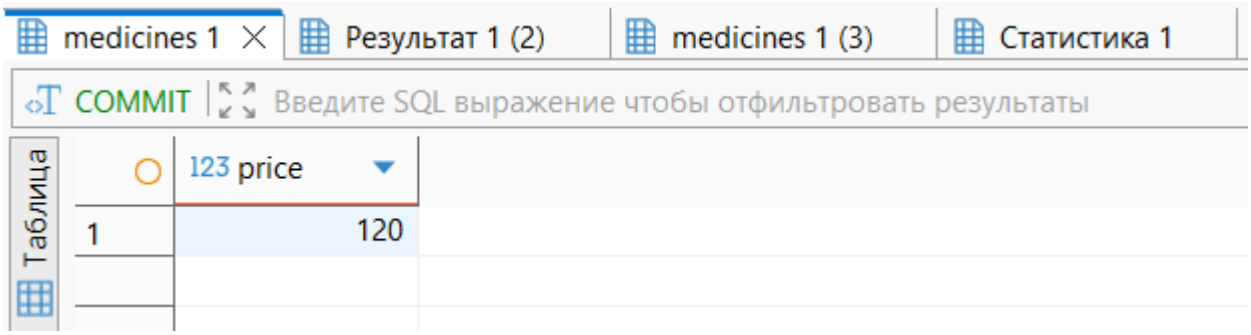


Рисунок 15 – Количество «Аспирина» при первом чтении

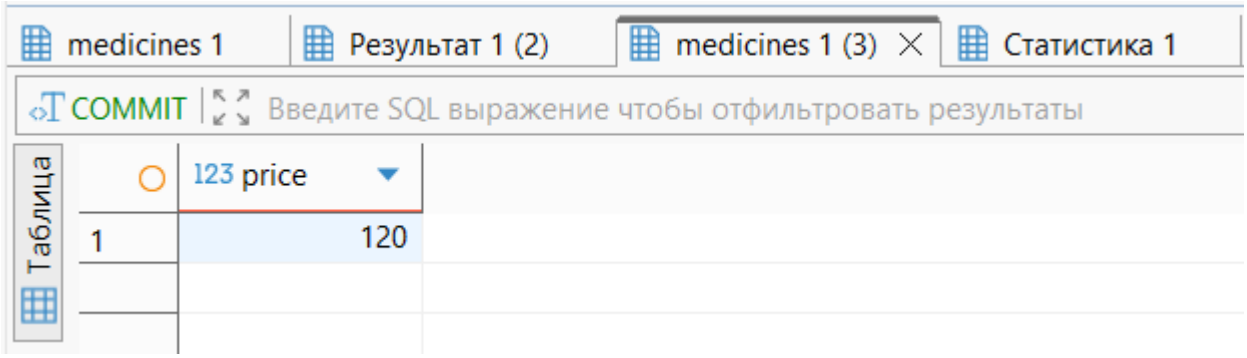


Рисунок 16 – Количество «Аспирина» при повторном чтении, спустя 10 секунд

Вывод:

Несмотря на то, что Скрипт Б успешно повысил цену до 12000.00, оба вывода демонстрируют одинаковое значение в 120.00.

Транзакция аудитора работала в изолированном «снимке» данных, и аномалия была предотвращена.

КОНТРОЛЬНЫЕ ВОПРОСЫ

1. Объясните своими словами, почему Seq Scan на большой таблице является неэффективной операцией.
2. Вы создали **INDEX** для ускорения **SELECT**, но это замедлит другие операции.
   Какие это операции (**INSERT**, **UPDATE**, **DELETE**) и почему?
3. Опишите бизнес-сценарий из вашей предметной области, где атомарность (принцип **ACID**) транзакции является **критически важной**.
4. В чём заключается аномалия «Неповторяемое чтение» *(Non-Repeatable Read)*?
5. Каким образом уровень изоляции **REPEATABLE READ** предотвращает эту аномалию, в отличие от **READ COMMITTED**?

КРАТКИЙ СПРАВОЧНЫЙ МАТЕРИАЛ

1. Оптимизация производительности (Индексы)

Индекс – это отдельная структура данных, связанная с таблицей и предназначенная для ускорения поиска данных.

План выполнения – это последовательность операций (план), которую оптимизатор СУБД выбирает для выполнения вашего SQL-запроса.

EXPLAIN ANALYZE – команда, которая выполняет запрос и показывает реальный план выполнения и реальное время выполнения.

- **Seq Scan (Sequential Scan)** – **Плохо.** Полный **перебор всей таблицы**.
- **Index Scan** – **Хорошо.** Быстрый **поиск по индексу**.

Основные типы *(методы доступа)* индексов в PostgreSQL:

B-Tree (Двоичное дерево) – используется по умолчанию, если не указан другой тип. Это самый распространенный и универсальный тип индекса.

Идеален для:

- операций сравнения (=, >, <, **BETWEEN**)
- сортировки (ORDER BY)
- поиска по шаблону (LIKE 'prefix%').

PostgreSQL также предлагает и другие методы доступа, например, GIN, GiST, Hash, BRIN, каждый из которых оптимизирован для своих специфических задач и типов данных

Особенности и стратегии индексирования

Часто стандартный метод доступа (например, B-Tree) используется в сочетании со специальными техниками, которые уточняют, что или какие данные попадают в индекс:

Индекс по выражению (Function-based) – индексирует не сами значения столбца, а результат функции или выражения, примененного к этим значениям.

Под капотом для этого чаще всего используется B-Tree (или другой выбранный тип).

```
CREATE INDEX ON table (LOWER(name));
```

Частичный (Partial) индекс – индексирует не все строки таблицы, а только то подмножество, которое отвечает определенному условию в блоке **WHERE**. Это позволяет создавать очень маленькие и быстрые индексы для часто используемых запросов.

```
CREATE INDEX ON table (id) WHERE archived = true;
```

2. Управление транзакциями (ACID)

Транзакция – это логическая, неделимая единица работы.

Принципы ACID:

- **A** ***(Atomicity / Атомарность)*** – «Всё или ничего».
- **C** ***(Consistency / Согласованность)*** – БД всегда остается в корректном состоянии.
- **I** ***(Isolation / Изолированность)*** – параллельные транзакции не мешают друг другу.
- **D** ***(Durability / Долговечность)*** – если COMMIT прошел, изменения навсегда сохранены.

Команды управления:

- `BEGIN;` – начать транзакцию.
- `COMMIT;` – успешно завершить и сохранить все изменения.
- `ROLLBACK;` – отменить все изменения с момента BEGIN.

Уровни изоляции (управляют аномалиями):

- `READ COMMITTED` (по умолчанию) – возможны «неповторяемые чтения». Каждый SELECT видит самые свежие зафиксированные данные.

- **REPEATABLE READ** – невозможны «неповторяемые чтения».
  Все **SELECT**ы внутри транзакции видят «снимок» данных, сделанный на момент начала транзакции.
  ```
  SET TRANSACTION ISOLATION LEVEL REPEATABLE READ;
  ```