# ПРАКТИЧЕСКАЯ РАБОТА 5.

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

По практической работе №5 формируется итоговый отчет, содержащий результат выполнения работы. Итоговый отчет должен содержать:

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

**Задание 1. Изучение транзакций COMMIT и ROLLBACK в PostgreSQL**

1. Запустите Query Tool (Редактор запросов) в PgAdmin и в окне запроса введите код, представленный на рисунке 1.1. Он содержит блок транзакции, который начинается с BEGIN, и оператор COMMIT, который завершает транзакцию и фиксирует в БД изменения, произведенные во время её выполнения:

```
 -- Посмотрим значение до изменения
SELECT 'Before' AS stage, "EMPLOYEE_ID", "FIRST_NAME", "LAST_NAME", "SALARY"
FROM "EmployeesDepartments"."EMPLOYEES"
WHERE "EMPLOYEE_ID" = 101;
-- Начинаем транзакцию
BEGIN;
-- Повышаем зарплату на 10% сотруднику с ID 101
UPDATE "EmployeesDepartments"."EMPLOYEES"
SET "SALARY" = "SALARY" * 1.1
WHERE "EMPLOYEE_ID" = 101;
-- Смотрим значение внутри транзакции (до коммита)
SELECT 'During' AS stage, "EMPLOYEE_ID", "FIRST_NAME", "LAST_NAME", "SALARY"
FROM "EmployeesDepartments"."EMPLOYEES"
WHERE "EMPLOYEE_ID" = 101;
-- Фиксация изменений
COMMIT;
-- Проверяем итоговое значение
SELECT 'After' AS stage, "EMPLOYEE_ID", "FIRST_NAME", "LAST_NAME", "SALARY"
FROM "EmployeesDepartments"."EMPLOYEES"
WHERE "EMPLOYEE_ID" = 101;
```

Рисунок 1.1 – Объявление блока транзакции и её фиксация

Обратите внимание - BEGIN начинает блок транзакции, то есть обозначает, что все операторы после команды BEGIN и до явной команды COMMIT или ROLLBACK будут выполняться в одной транзакции. По умолчанию (без BEGIN) pgAdmin выполняет транзакции

в режиме «autocommit» (автофиксация), то есть каждый оператор выполняется в своей отдельной транзакции, которая неявно фиксируется в конце оператора (если оператор был выполнен успешно; в противном случае транзакция откатывается).

Для отключения режима автокоммита в pgAdmin можно воспользоваться настройками приложения. Откройте меню Файл (File) → Свойства (Preferences) → Инструменты запроса (Query Tools) → Свойства (Query Tool) и отключите переключатели Autocommit и Auto-rollback. Эти изменения будут применены ко всем новым сессиям, запускаемым в pgAdmin.

Если требуется отключить автокоммит только для текущей сессии, можно использовать один из двух способов:

- Ручное отключение. В окне инструмента запросов нажмите на стрелку рядом с кнопкой «Выполнить» (Execute) и снимите галочку «Включить автокоммит» (Enable autocommit).
- Отключение через команду. В окне Query Tool введите и выполните команду:

SET autocommit = off;

После этого каждая команда DML будет выполняться в рамках общей транзакции. Чтобы завершить транзакцию, необходимо явно выполнить COMMIT или ROLLBACK. В блоке транзакции операторы выполняются быстрее, так как для запуска/фиксации транзакции производится масса операций, нагружающих процессор и диск. Кроме того, выполнение нескольких операторов в одной транзакции позволяет обеспечить целостность при внесении серии связанных изменений; другие сеансы не видят промежуточное состояние, когда произошли ещё не все связанные изменения.

2. Выполните код последовательно, вы увидите значения до, во время и после транзакции, зафиксируйте результаты в отчете.

3. Теперь исследуем поведение отката транзакции ROLLBACK. Введите код, представленный на рисунке 1.2. Выполните его последовательно, зафиксировав в отчете результаты каждой инструкции.

```
-- Значения до изменений
SELECT 'Before' AS stage, "EMPLOYEE_ID", "FIRST_NAME", "LAST_NAME", "SALARY"
FROM "EmployeesDepartments"."EMPLOYEES"
WHERE "EMPLOYEE_ID" <= 103
ORDER BY 2;
-- Начинаем транзакцию
BEGIN;
-- Повышаем зарплату всем сотрудникам до ID 103
UPDATE "EmployeesDepartments"."EMPLOYEES"
SET "SALARY" = "SALARY" * 1.1
WHERE "EMPLOYEE_ID" <= 103;
-- Значения внутри транзакции, до отката
SELECT 'Within transaction' AS stage, "EMPLOYEE_ID", "FIRST_NAME", "LAST_NAME", "SALARY"
FROM "EmployeesDepartments"."EMPLOYEES"
WHERE "EMPLOYEE_ID" <= 103
ORDER BY 2;
-- Откат транзакции
ROLLBACK;
-- Проверяем, что изменения не сохранены
SELECT 'After' AS stage, "EMPLOYEE_ID", "FIRST_NAME", "LAST_NAME", "SALARY"
FROM "EmployeesDepartments"."EMPLOYEES"
WHERE "EMPLOYEE_ID" <= 103
ORDER BY 2;
```

Рисунок 1.2 – Откат транзакции

4. Изучите внимательно результаты, они показывают, что произошло. Зафиксируйте их в отчете.

5. Изучим создание точки сохранения внутри текущей транзакции. Оператор SAVEPOINT используется для создания точки сохранения внутри текущей транзакции, что позволяет откатить изменения только до этой точки, а не всю транзакцию целиком. Это полезно в сценариях, когда нужно выполнить несколько последовательных операций в рамках одной транзакции, но есть риск, что одна из них может вызвать ошибку. Если ошибка происходит, можно откатиться до определенной точки, исправить проблему и продолжить, не теряя все предыдущие изменения. Чтобы понять, как работает SAVEPOINT, введите код, представленный на рисунке 1.3, и выполните его поэтапно.

```
BEGIN;
SELECT 'Trans started' AS stage, "EMPLOYEE_ID", "FIRST_NAME", "LAST_NAME", "SALARY"
FROM "EmployeesDepartments"."EMPLOYEES"
WHERE "EMPLOYEE_ID" = 104;

-- Первая точка сохранения с именем sp1
SAVEPOINT sp1;
UPDATE "EmployeesDepartments"."EMPLOYEES"
SET "SALARY" = 3000
WHERE "EMPLOYEE_ID" = 104;

SELECT 'After SAVEP sp1' AS stage, "EMPLOYEE_ID", "FIRST_NAME", "LAST_NAME","SALARY"
FROM "EmployeesDepartments"."EMPLOYEES"
WHERE "EMPLOYEE_ID" = 104;

-- Вторая точка сохранения с именем sp2
SAVEPOINT sp2;
UPDATE "EmployeesDepartments"."EMPLOYEES"
SET "SALARY" = 10000
WHERE "EMPLOYEE_ID" = 104;
SELECT 'After SAVEP sp2' AS stage, "EMPLOYEE_ID", "FIRST_NAME", "LAST_NAME","SALARY"
FROM "EmployeesDepartments"."EMPLOYEES"
WHERE "EMPLOYEE_ID" = 104;

-- Откат только до второго savepoint
ROLLBACK TO SAVEPOINT sp2;

SELECT 'After ROLLB sp2' AS stage, "EMPLOYEE_ID", "FIRST_NAME", "LAST_NAME","SALARY"
FROM "EmployeesDepartments"."EMPLOYEES"
WHERE "EMPLOYEE_ID" = 104;

UPDATE "EmployeesDepartments"."EMPLOYEES"
SET "SALARY" = "SALARY"+100
WHERE "EMPLOYEE_ID" = 104;
-- Фиксация полной транзакции
COMMIT;

SELECT 'After trans' AS stage, "EMPLOYEE_ID", "FIRST_NAME", "LAST_NAME", "SALARY"
FROM "EmployeesDepartments"."EMPLOYEES"
WHERE "EMPLOYEE_ID" = 104;
```

Рисунок 1.3 – Установка точек сохранения внутри транзакции

**Примечание:** Для отката к установленной точке сохранения предназначена команда ROLLBACK TO. Точки сохранения могут быть установлены только внутри блока транзакции. В одной транзакции можно определить несколько точек сохранения.

6. Изучите внимательно результаты, они показывают, что произошло. Зафиксируйте их в отчете.

## Задание 2. Поиск и обнаружение блокировок

1. PostgreSQL использует блокировки для обеспечения корректности и изоляции транзакций. В этом задании вы научитесь: создавать ситуацию блокировки, анализировать блокировки в системных представлениях, выявлять блокирующие и ожидающие процессы. Для этого подготовим тестовый пример. В окне редактора запроса (Query Tool) введите код, представленный на рисунке 2.1.

```
DROP TABLE IF EXISTS public.t1;

CREATE TABLE public.t1 (
    id    int PRIMARY KEY,
    price numeric(10,2)
);

INSERT INTO public.t1 VALUES
(1, 10.00),
(2, 20.00),
(3, 30.00);
```

Рисунок 2.1 – Создание тестовой таблицы и наполнение её данными

2. Чтобы увидеть блокировки в действии, нам понадобятся три активные сессии к одной и той же БД. Для их создания в дереве объектов (Object Explorer) нажмите правой кнопкой мыши на название базы данных или на узел "Query Tools" под ней. Выберите "Query Tool" в контекстном меню. Откроется первая сессия/вкладка. Повторите выбор "Query Tool" в контекстном меню еще два раза. Должны появиться 3 три отдельные вкладки редактора запросов, каждая из которых представляет собой независимую сессию к вашей БД.

3. Для удобства дальнейшей работы переименуйте созданные вкладки. Для этого нажмите правой кнопкой мыши по заголовку нужной вкладки и выберите пункт «Переименовать» (рисунок 2.2). Задайте вкладкам имена Connection1, Connection2 и Connection3 соответственно.

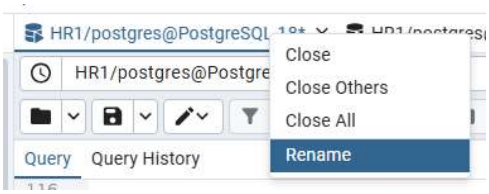


Рисунок 2.2 – Переименование вкладки редактора запросов

Для проверки, что все три сессии подключены к нужной базе, выполните: запрос: SELECT current_database();

4. В первой вкладке (Connection1) выполните код (рисунок 2.3) для обновления строки в таблице public.t1, которое заключается в увеличении текущей цены на 1 для товара с идентификатором 2.

```
BEGIN;

UPDATE public.t1
SET price = price + 1.00
WHERE id = 2;

SELECT 'Inside Connection 1' AS stage, id, price
FROM public.t1
WHERE id = 2;
```

Рисунок 2.3 – Транзакция в первой сессии

**Примечание.** Для обновления строки сеанс должен был установить монопольную блокировку на уровне строки, и, если обновление прошло успешно, значит, блокировка была установлена. Монопольные блокировки сохраняются до конца транзакции, а поскольку транзакция остается открытой, блокировка продолжает удерживаться.

5. Перейдите во вкладку Connection2 и выполните код (рисунок 2.4), чтобы попытаться обновить ту же самую строку.

```
BEGIN;
SELECT id, price
FROM public.t1
WHERE id = 2;

UPDATE public.t1
SET price = 0
WHERE id = 2;

SELECT id, price
FROM public.t1
WHERE id = 2;
```

Рисунок 2.4 – Чтение во второй сессии

**Примечание.** Первый SELECT может выполниться сразу, потому что обычное чтение не требует блокировки на уровне строки. Но при попытке UPDATE вторая сессия попытается получить монопольную блокировку строки, которая несовместима с блокировкой, удерживаемой в Connection1. Поэтому Connection2 будет ждать, пока Connection1 не завершит транзакцию.

6. Чтобы увидеть, какие именно блокировки удерживаются активными транзакциями, а также какая сессия ожидает блокировку, необходимо воспользоваться системными представлениями PostgreSQL — pg_locks и pg_stat_activity. Эти представления позволяют в реальном времени наблюдать: какие объекты заблокированы, какой тип блокировки удерживается, какая сессия ждёт блокировку, какой SQL-запрос в данный момент выполняется, состояние транзакций. Чтобы прояснить подобную ситуацию, в окне Connection3 выполните запрос к представлению pg_stat_activity (рисунок 2.5), которое предоставляет информацию о текущих сеансах базы данных, выполняемых запросах, состоянии транзакций и позволяет увидеть, какая сессия блокирует другую.

```sql
SELECT pid,
       usename,
       application_name,
       state,
       wait_event_type,
       wait_event,
       query
FROM pg_stat_activity
WHERE datname = current_database()
ORDER BY pid;
```

Рисунок 2.5 – Обращение к представлению pg_stat_activity

**Примечание.** Каждый сеанс в PostgreSQL имеет собственный идентификатор серверного процесса (pid), по которому можно понять, какой именно backend удерживает блокировку или ожидает её освобождения. Поле *usename* показывает имя пользователя, выполняющего запрос, а *application_name* — имя клиента, из которого установлено подключение (в случае pgAdmin обычно отображается pgAdmin Query Tool). Поле *state* отражает текущее состояние сеанса: например, *active* означает, что в данный момент выполняется SQL-команда; *idle* говорит о том, что сеанс не выполняет запросов и не находится внутри транзакции; состояние *idle in transaction* указывает, что транзакция была начата, но после последней команды клиент ничего не отправляет, при этом транзакция остаётся открытой и удерживает блокировки; состояние *idle in transaction (aborted)* означает, что внутри открытой транзакции произошла ошибка, и до выполнения ROLLBACK никакие новые команды не будут обработаны. Пара полей *wait_event_type* и *wait_event* описывает, чего именно ожидает процесс: например, события вида *Lock* говорят о попытке получить блокировку, а *tuple* или *relation* указывают, какого типа блокировка ожидается. Поле *query* содержит последний выполненный или текущий SQL-запрос, что позволяет понять, какая именно операция привела сеанс к ожиданию или удержанию блокировки. Такой набор информации делает возможным сопоставление наблюдаемых в Connection1 и Connection2 ситуаций с конкретными backend-процессами в *pg_stat_activity*, что особенно полезно при анализе блокировок и зависших транзакций.

7. Теперь поработаем с представлением pg_locks, которое позволяет увидеть все блокировки, удерживаемые и ожидаемые в текущей базе данных. Выполните следующий запрос: (рисунок 2.6).

```sql
SELECT
    l.pid,
    a.application_name,
    a.state,
    l.locktype,
    l.relation::regclass AS locked_relation,
    l.page,
    l.tuple,
    l.virtualxid,
    l.transactionid,
    l.mode,
    l.granted
FROM pg_locks l
LEFT JOIN pg_stat_activity a ON a.pid = l.pid
WHERE a.datname = current_database()
ORDER BY l.pid, l.locktype;
```

Рисунок 2.6 – Обращение к представлению pg_locks

Этот запрос выводит сводную информацию сразу из двух системных представлений: pg_locks, где фиксируются сведения о блокировках, и pg_stat_activity, отображающего состояние серверных процессов. Таким образом можно увидеть, какой процесс удерживает или ожидает конкретную блокировку, какое именно действие он выполняет и в каком состоянии сейчас находится. Поле locktype показывает тип блокировки (к примеру, relation, tuple, transactionid), а поле locked_relation позволяет понять, к какой таблице относится блокировка. Параметры page и tuple заполняются для блокировок уровня строки или страницы, а mode указывает запрашиваемый режим блокировки, например, AccessShareLock или ExclusiveLock. Отдельно важно поле granted, которое показывает, удерживается ли блокировка сейчас или процесс только ожидает её получения. Поля из представления pg_stat_activity, такие как application_name и state, помогают сопоставить блокировку с конкретной сессией — например, увидеть, что именно та вкладка, в которой вы выполняли UPDATE, находится в состоянии *idle in transaction* и удерживает монопольную блокировку.

Используя такой запрос, легко определить, какая сессия заблокировала строку или таблицу, кто ожидает снятия блокировки и какие именно ресурсы участвуют в конфликте. Это особенно полезно при отладке многопользовательских транзакций, поиске взаимных блокировок и анализе поведения вашего приложения в условиях конкурентного доступа.

Если всё сделано правильно, то Connection1 удерживает монопольную блокировку записи: mode = RowExclusiveLock; granted = true. А Connection2 пытается читать или обновлять ту же строку: granted = false— значит, стояние в ожидании блокировки.

8. Для завершения транзакций и наблюдение за эффектом вернитесь в Connection1 и выполните: COMMIT;

После коммита блокировка снимается — и Connection2 сможет продолжить выполнение (его UPDATE завершится). Зафиксируйте эту транзакцию, вызвав оператор commit и проверьте результат, цена должна обнулиться.

9. Снова выполните запросы к pg_locks и pg_stat_activity, чтобы убедиться, что блокировки исчезли.

**Задание 3. Уровени изоляции READ UNCOMMITTED и READ COMMITTED**

Уровни изоляции определяют поведение параллельно работающих пользователей, читающих и записывающих данные. Читающий процесс – это любая инструкция, извлекающая данные и применяющая по умолчанию совместную блокировку. Пишущая инструкция - это любая инструкция, модифицирующая таблицу и запрашивающая монопольную блокировку. В этом задании вы научитесь задавать уровни изоляции транзакций и посмотрите, как они работают.

Уровень READ UNCOMMITTED является самым низким уровнем изоляции транзакций. При его использовании транзакция может видеть изменения, внесённые другими транзакциями, даже если эти изменения ещё не были зафиксированы. Такое поведение позволяет читать так называемые «грязные» данные — значения, которые впоследствии могут быть отменены.

Однако в PostgreSQL данный уровень изоляции фактически не реализован. Это связано с тем, что PostgreSQL использует механизм MVCC (Multiversion Concurrency Control) — многоверсионный контроль параллелизма. Благодаря MVCC каждая транзакция работает со снимком данных, актуальным на момент её начала, и видит только завершённые, зафиксированные изменения. Поэтому режим READ UNCOMMITTED в PostgreSQL не применим.

Вместо него PostgreSQL автоматически использует уровень READ COMMITTED, который предотвращает появление грязных чтений, так как каждая операция чтения получает только зафиксированные данные. Этот режим является стандартным и подходит для большинства приложений.

1. Для демонстрации поведения транзакций в режиме READ COMMITTED нам понадобятся две активные сессии работы с базой данных. Выберите "Query Tool" в контекстном меню. Откроется первая сессия/вкладка, назовем ее Session1. Повторный выбор "Query Tool" позволит открыть вторую сессию – Session2.

2. В окне Session1 запустите транзакцию и выполните код, аналогичный коду предыдущего задания для обновления строки в таблице t1, которое заключается в установлении текущей цены 100 для товара 3 (3.1):

```
BEGIN;

SELECT id, price
FROM public.t1
WHERE id = 3;

UPDATE public.t1
SET price = 100
WHERE id = 3;

SELECT id, price
FROM public.t1
WHERE id = 3;
```

Рисунок 3.1 – Запуск транзакции в Session1

**Примечание:** Вы видите, что обновление прошло успешно, но транзакция остается открытой. Это означает монопольную блокировку строки в подключении Session1.

3. В окне Session2 установите уровень изоляции READ COMMITTED и выполните SELECT, для чего введите и выполните следующий код (рисунок 3.2):

```
BEGIN ISOLATION LEVEL READ COMMITTED;
SELECT id, price
FROM public.t1
WHERE id = 3;
```

Рисунок 3.2 – Установка уровня изоляции READ COMMITTED в Session2

Вы увидите старое значение, так как изменения в Session1 еще не зафиксированы.

**Примечание:** Уровень изоляции READ COMMITTED используется по умолчанию, но его можно явно указать, как это сделано в нашем случае.

4. Теперь в окне Session1 выполните фиксацию транзакции – COMMIT;

5. Откройте окно Session2 и в том же блоке транзакции выполните SELECT еще раз. Теперь вы увидите новое зафиксированное значение, потому что уровень READ COMMITTED позволяет каждому новому оператору SELECT видеть все данные, зафиксированные на этот момент. Завершите транзакцию в окне Session2, выполнив COMMIT или ROLLBACK.

**Примечание:** в режиме READ COMMITTED разные операторы SELECT в рамках одной транзакции могут видеть разные данные, если между их выполнениями другая транзакция зафиксировала изменения.

**Задание 4. Уровень изоляции REPEATABLE READ**

Уровень изоляции REPEATABLE READ в PostgreSQL гарантирует, что внутри одной транзакции все повторные чтения вернут данные из одного и того же «снимка» (snapshot) базы данных. Это означает, что даже если параллельная транзакция изменяет существующие строки или добавляет новые, первая транзакция не увидит никаких изменений, произошедших после начала её работы. Благодаря этому предотвращается не только неповторяемое чтение (ситуация, когда одна и та же строка при повторном запросе возвращает разные значения в рамках одной транзакции), но и фантомное чтение (ситуация, когда параллельная транзакция добавляет новые строки, подходящие под условие запроса. При повторном выполнении запроса внутри первой транзакции эти новые строки могут появиться как «фантомы».).

Таким образом за счет использования снимков данных изоляция в PostgreSQL получается строже, чем того требует стандарт, поэтому уровень Repeatable Read в PostgreSQL не допускает не только неповторяющегося, но и фантомного чтения.

1. Сначала рассмотрим пример неповторяемого чтения. Для этого воспользуемся созданными нами сессиями Session1 и Session2. В окне Session1 запустите транзакцию и выполните код, аналогичный коду предыдущего задания для чтения данных о товаре 3 (рисунок 4.1):

```
BEGIN ISOLATION LEVEL REPEATABLE READ;

SELECT id, price
FROM public.t1
WHERE id = 3;
```

Рисунок 4.1 – Запуск транзакции с уровнем изоляции REPEATABLE READ

**Примечание:** Код вернет текущую цену товара, но подключение Session1 все еще удерживает совместную блокировку строки 3 в таблице.

2. В окне Session2 выполните следующий код (рисунок 4.2), чтобы попытаться изменить цену товара 3.

```
BEGIN;
UPDATE public.t1
SET price = 1
WHERE id = 3;
COMMIT;
```

Рисунок 4.2 – Изменение цены товара 3 в сессии Session2

3. В окне Session1 еще раз считайте данные (они не изменились!) и завершите транзакцию (рисунок 4.3).

```
SELECT id, price
FROM public.t1
WHERE id = 3;

COMMIT;
```

Рисунок 4.3 – Выполнение повторного запроса и завершение транзакции в Session1

**Примечание:** При повторном выполнении запроса в Session1 возвращается та же строка, что и при первом чтении. Это происходит потому, что уровень изоляции REPEATABLE READ фиксирует для транзакции снимок данных на момент её начала. Транзакция в Session2 была создана позже, поэтому любые её изменения, даже после фиксации, не могут повлиять на результат чтения в Session1. Session1 продолжает работать с тем состоянием базы данных, которое существовало до начала Session2.

4. Рассмотрим пример фантомного чтения, чтобы на практике убедиться, как PostgreSQL обрабатывает новые строки в рамках уровня изоляции REPEATABLE READ. В окне Session1 запустите транзакцию и выполните код, аналогичный коду предыдущего задания для чтения данных о товаре с ценой 1(рисунок 4.4):

```
BEGIN ISOLATION LEVEL REPEATABLE READ;

SELECT id, price
FROM public.t1
WHERE price = 1;
```

Рисунок 4.4 – Запуск транзакции с уровнем изоляции REPEATABLE READ

5. В окне Session2 выполните следующий код (рисунок 4.5), чтобы добавить ещё один товар с идентификатором 4 и ценой 1.

```
BEGIN;
INSERT INTO public.t1
VALUES (4, 1);
COMMIT;
```

Рисунок 4.5 – Добавление новой строки в Session2

6. В окне Session1 еще раз считайте данные (они не изменились!) и завершите транзакцию (рисунок 4.6).

```
SELECT id, price
FROM public.t1
WHERE price = 1;

COMMIT;
```

Рисунок 4.6 – Выполнение повторного запроса и завершение транзакции в Session1

**Примечание:** в СУБД PostgreSQL фантомное чтение не возникает: повторные запросы внутри одной транзакции видят один и тот же набор строк, даже если параллельная транзакция добавляет новые строки, подходящие под условие запроса. Это связано с особенностями реализации уровня изоляции REPEATABLE READ в PostgreSQL: каждая транзакция работает со снимком базы данных, зафиксированным на момент её начала.

## Задание 5. Уровень изоляции SERIALIZABLE

Уровень изоляции SERIALIZABLE в PostgreSQL обеспечивает максимальную изоляцию транзакций, гарантируя, что результаты выполнения параллельных транзакций эквивалентны некоторому последовательному (последовательно выполняемому) порядку их выполнения. В отличие от классического понимания уровня SERIALIZABLE, который предполагает строгие блокировки для предотвращения любых конфликтов,

PostgreSQL использует MVCC и проверку конфликтов при фиксации транзакции. Это означает, что транзакции могут свободно читать и изменять строки, но при попытке фиксации PostgreSQL проверяет, не возникли ли логические конфликты с другими параллельными транзакциями.

Логический конфликт возникает, если изменения, внесённые одной транзакцией, потенциально нарушают последовательность результатов другой транзакции — например, когда две транзакции одновременно пытаются обновить или удалить одну и ту же строку, либо создают условия, при которых результат последовательного выполнения был бы другим. В случае обнаружения такого конфликта одна из транзакций будет отменена с ошибкой serialization failure, что требует её повторного выполнения.

Благодаря этому достигается строгий уровень изоляции без необходимости блокировки всей таблицы, что позволяет одновременно обеспечивать отсутствие неповторяемого и фантомного чтения, сохраняя высокую параллельность работы с базой данных.

1. Для демонстрации поведения транзакций в режиме SERIALIZABLE нам понадобятся две активные сессии работы с базой данных. Для этого воспользуемся созданными нами сессиями Session1 и Session2. В окне Session1 запустите транзакцию и выполните код, аналогичный коду предыдущего задания для чтения данных о товарах с ценой 1 (рисунок 5.1).

```
BEGIN ISOLATION LEVEL SERIALIZABLE;

-- обновиляем строку с id = 1
UPDATE public.t1
SET price = 111
WHERE id = 1;

-- Читаем строку с id = 1
SELECT id, price
FROM public.t1
WHERE id = 1;
```

Рисунок 5.1 – Запуск транзакции в режиме SERIALIZABLE в Session1

**Примечание:** Вы должны получить обновленную цену для товара 1.

2. В окне Session2 попытайтесь также изменить цену товара 1, для этого выполните код (рисунок 5.2).

```
BEGIN ISOLATION LEVEL SERIALIZABLE;

UPDATE public.t1
SET price = 222
WHERE id = 1;

COMMIT;
```

Рисунок 5.2 – Обновление цены товара в Session2

**Примечание:** Транзакция в Session2 ожидает закрытия транзакции в Session1 так как присутствует совместная блокировка строки.

3. Вернитесь в окно Session1, снова выполните извлечение данных и зафиксируйте транзакцию (COMMIT). После фиксации транзакции в Session1 блокировки на обновляемые данные снимаются, и логично ожидать, что транзакция в Session2 сможет завершиться.

Однако в рамках уровня изоляции SERIALIZABLE повторное обновление одних и тех же данных в параллельных транзакциях запрещено. Поэтому при попытке выполнения транзакции в Session2 возникает ошибка (см. рисунок 5.3). Это является главным отличием SERIALIZABLE от READ COMMITTED. Все изменения, сделанные транзакцией Session2 до ошибки, будут автоматически отменены.

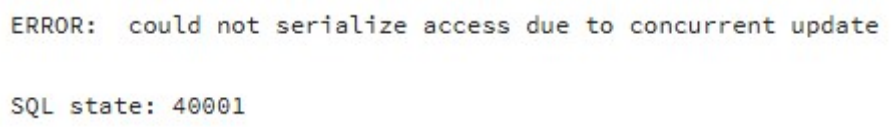


Рисунок 5.3 – Ошибка из-за уровня изоляции SERIALIZABLE.