Отчет по лабораторной работе №3

Модель многопользовательского доступа: MVCC

Дата: 2025-10-05
Семестр: 4 курс 1 полугодие – 7 семестр
Группа: ПИЖ-б-о-22-1
Дисциплина: Администрирование баз данных
Студент: Душин Александр Владимирович

Цель работы

Изучить принципы многоверсионного управления конкурентным доступом (MVCC) в PostgreSQL. Получить практические навыки наблюдения за работой MVCC, анализа версий строк, снимков данных и уровней изоляции транзакций. Освоить использование расширений и системных представлений для исследования внутренней структуры данных.

Теоретическая часть

MVCC (Multiversion Concurrency Control) — механизм, позволяющий нескольким транзакциям работать с одними и теми же данными одновременно, минимизируя блокировки. Каждая транзакция видит согласованный «снимок» данных на момент своего начала.

1. Версии строк: При изменении строки создается ее новая версия. Старая версия остается в таблице до очистки.
2. Системные поля:

- xmin – идентификатор транзакции, создавшей версию строки.
- xmax — идентификатор транзакции, удалившей версию строки (или заблокировавшей ее для обновления).
- ctid — физическое расположение версии строки в таблице (номер страницы и позиции в ней).

3. Уровни изоляции: Определяют, какие аномалии параллелизма допустимы:

- Read Committed (По умолчанию): Видны только зафиксированные данные. Возможны неповторяемое чтение и фантомное чтение.
- Repeatable Read: Гарантирует, что данные, прочитанные в транзакции, не изменятся. Предотвращает неповторяемое чтение, возможны фантомы.
- Serializable: Самый строгий уровень, предотвращает все аномалии.

4. Снимок данных (Snapshot): Набор идентификаторов транзакций, активных на момент начала текущей транзакции. Определяет, какие версии строк видимы текущей транзакции.

Практическая часть

Часть 1. Уровни изоляции и аномалии

Выполненные задачи:

1. Создал таблицу `iso_test (id INT, data TEXT)` и вставил одну строку. В сеансе 1 начал транзакцию с уровнем `READ COMMITTED` и выполнил `SELECT iso_test;. * FROM iso_test;`. В сеансе 2 удалил строку и зафиксировал изменения (`DELETE ...; COMMIT;`). В сеансе 1 выполнил тот же `SELECT` повторно. Вывело 0 строк. Завершил транзакцию в сеансе 1.

2. Повторил предыдущий эксперимент, но в сеансе 1 начал транзакцию с `BEGIN ISOLATION LEVEL REPEATABLE READ;`. `READ COMMITTED` берёт новый снимок на каждый запрос. `REPEATABLE READ` фиксирует снимок данных и держит его неизменным на всю транзакцию.

3. В сеансе 1 начал транзакцию и создал новую таблицу `new_table`, вставил в нее строку. Не фиксировал. В сеансе 2 выполнил `SELECT * FROM new_table;`. Выдало ошибку о несуществовании таблицы. Зафиксировал транзакцию в сеансе 1. Повторил запрос в сеансе 2, таблица появилась. Повторил процесс, но вместо фиксации откатил транзакцию в сеансе 1. После `ROLLBACK;` таблица также нету в сеансе 2.

4. В сеансе 1 начал транзакцию и выполнил `SELECT * FROM iso_test;`. Попытался в сеансе 2 выполнить `DROP TABLE iso_test;`. Удалить таблицу не получится, пока не завершим транзакцию в первом сеансе.

Команды:

Задача 1:

```
-- Сеанс 1
CREATE DATABASE lab03_db;
\c lab03_db
CREATE TABLE iso_test(id INT, data TEXT);
INSERT INTO iso_test VALUES (1,'row1');

BEGIN ISOLATION LEVEL READ COMMITTED;
SELECT * FROM iso_test;
```

```
-- Сеанс 2
\c lab03_db
DELETE FROM iso_test WHERE id=1;
COMMIT;
```

```
-- Сеанс 1
SELECT * FROM iso_test;
COMMIT;
```

Задача 2:

```
-- Сеанс 1
INSERT INTO iso_test VALUES (1,'row1');
BEGIN ISOLATION LEVEL REPEATABLE READ;
SELECT * FROM iso_test;
```

```
-- Сеанс 2
DELETE FROM iso_test WHERE id=1;
COMMIT;
```

```
-- Сеанс 1
SELECT * FROM iso_test;
COMMIT;
```

Задача 3:

```
-- Сеанс 1
BEGIN;
CREATE TABLE new_table(id int);
INSERT INTO new_table VALUES (10);
```

```
-- Сеанс 2
SELECT * FROM new_table;
```

```
-- Сеанс 1
COMMIT;
```

```
-- Сеанс 2
SELECT * FROM new_table;
```

```
-- Сеанс 1
DROP TABLE new_table;
BEGIN;
CREATE TABLE new_table(id int);
INSERT INTO new_table VALUES (10);
```

```
-- Сеанс 2
SELECT * FROM new_table;
```

```
-- Сеанс 1
ROLLBACK;
```

```
-- Сеанс 2
SELECT * FROM new_table;
```

Задача 4:

```
-- Сеанс 1
BEGIN;
SELECT * FROM iso_test;
```

```
-- Сеанс 2
DROP TABLE iso_test;
```

```
-- Сеанс 1
COMMIT;
```

Фрагменты вывода:

Задача 1:

```
CREATE DATABASE
You are now connected to database "lab03_db" as user "postgres".

CREATE TABLE

INSERT 0 1

BEGIN

id | data
----+------
  1 | row1
(1 row)
```

```
You are now connected to database "lab03_db" as user "postgres".

DELETE 1

WARNING:  there is no transaction in progress
COMMIT
```

```
 id | data
----+------
(0 rows)

COMMIT
```

Задача 2:

```
INSERT 0 1

BEGIN

id | data
----+------
  1 | row1
(1 row)
```

```
DELETE 1

WARNING:  there is no transaction in progress
COMMIT
```

```
 id | data
----+------
  1 | row1
(1 row)

COMMIT
```

Задача 3:

```
BEGIN
CREATE TABLE
INSERT 0 1
```

```
ERROR:  relation "new_table" does not exist
LINE 1: SELECT * FROM new_table;
```

```
COMMIT
```

```
 id
----
 10
(1 row)
```

```
DROP TABLE
BEGIN
CREATE TABLE
INSERT 0 1
```

```
ERROR:  relation "new_table" does not exist
LINE 1: SELECT * FROM new_table;
                      ^
```

```
ROLLBACK
```

```
ERROR:  relation "new_table" does not exist
LINE 1: SELECT * FROM new_table;
                      ^
```

Задача 4:

```
-- Сеанс 1
BEGIN
 id |   data
----+-----------
```

```
  1 | test data
(1 row)
```

```
-- Сеанс 1
COMMIT
```

```
-- Сеанс 2
DROP TABLE
```

Часть 2. Фантомное чтение и снимки

Выполненные задачи:

1. Создал пустую таблицу `phantom_test (id INT)`. Продемонстрировал на уровне `Read Committed`, что аномалия "фантомное чтение" не предотвращается (вставка новых строк в другом сеансе становится видимой).
2. В сеансе 1 начал транзакцию с уровнем `Repeatable Read` (пока без запросов). В сеансе 2 удалил все строки из `phantom_test` и зафиксировал. В сеансе 1 выполнил `SELECT * FROM phantom_test;`. Удалённые стркои вывелись. Выполнил в сеансе 1 запрос `SELECT * FROM pg_database;` (не касаясь `phantom_test`). При последующем запросе также вывело удалённые строки.
3. Убедился, что `DROP TABLE` является транзакционной операцией (можно откатить).

Команды:

Задача 1:

```
-- Сеанс 1
CREATE TABLE phantom_test(id INT);
BEGIN ISOLATION LEVEL READ COMMITTED;
SELECT count(*) FROM phantom_test;
```

```
-- Сеанс 2
INSERT INTO phantom_test VALUES (1),(2),(3);
COMMIT;
```

```
-- Сеанс 1
SELECT count(*) FROM phantom_test;
COMMIT;
```

Задача 2:

```
-- Сеанс 1
BEGIN ISOLATION LEVEL REPEATABLE READ;
```

```
-- Сеанс 2
DELETE FROM phantom_test;
COMMIT;
```

```
-- Сеанс 1
SELECT * FROM phantom_test;
SELECT * FROM pg_database;
SELECT * FROM phantom_test;
COMMIT;
```

Задача 3:

```
CREATE TABLE test_ddl (id INT);
INSERT INTO test_ddl VALUES (1);

BEGIN;
DROP TABLE test_ddl;
SELECT * FROM test_ddl;

ROLLBACK;
SELECT * FROM test_ddl;
```

Фрагменты вывода:

Задача 1:

```
-- Сеанс 1
CREATE TABLE

BEGIN

 count
-------
     0
(1 row)
```

```
-- Сеанс 2
INSERT 0 3

WARNING:  there is no transaction in progress
COMMIT
```

```
-- Сеанс 1
 count
-------
     3
(1 row)

COMMIT
```

Задача 2:

```
-- Сеанс 1
BEGIN
```

```
-- Сеанс 2
DELETE 3

WARNING:  there is no transaction in progress
COMMIT
```

```
 id
----
  1
  2
  3
(3 rows)

 id
----
  1
  2
  3
(3 rows)
```

Задача 3:

```
CREATE TABLE

INSERT 0 1

BEGIN

DROP TABLE

ERROR:  relation "test_ddl" does not exist

ROLLBACK

 id
----
  1
(1 row)
```

Часть 3. Версии строк и pageinspect

Выполненные задачи:

1. Создал таблицу `version_test (id INT)`. Вставил одну строку. Дважды обновил эту строку (`UPDATE ...`), а затем удалил её (`DELETE`). Используя расширение `pageinspect` (`heap_page_items`), сейчас версий строк 3. Все версии погашены, так как t_xmax не равен 0. По t_ctid видно, что первая версия строки ссылается на вторую версия, так как вторая удалила первую версию. Вторая версия переадресовывает на третью версию. Так как это была последняя версия строки она ссылается на саму себя.

2. Определил, в какой странице (блоке) находится строка в `pg_class`, описывающая саму таблицу `pg_class`. Используя `pageinspect`, Актуальных версий строк 2, так как только у двух строк t_xmax = 0.

3. Включил в `psql` параметр `ON_ERROR_ROLLBACK`. Создал ситуацию с ошибкой в транзакции и убедился, что этот режим использует точки сохранения (`SAVEPOINT`), позволяя продолжить работу транзакции после ошибки.

Команды:

Задача 1:

```
CREATE TABLE version_test (id INT);
INSERT INTO version_test VALUES (1);

SELECT ctid, xmin, xmax, * FROM version_test;

UPDATE version_test SET id = 2;
SELECT ctid, xmin, xmax, * FROM version_test;

UPDATE version_test SET id = 3;
```

```
SELECT ctid, xmin, xmax, * FROM version_test;

DELETE FROM version_test;
SELECT ctid, xmin, xmax, * FROM version_test;

CREATE EXTENSION pageinspect;
SELECT * FROM heap_page_items(get_raw_page('version_test', 0));
```

Задача 2:

```
SELECT ctid FROM pg_class WHERE relname = 'pg_class';
```

```
SELECT lp, t_xmin, t_xmax, t_ctid
FROM heap_page_items(get_raw_page('pg_class', 4))
WHERE lp_flags = 1;
```

Задача 3:

```
\set ON_ERROR_ROLLBACK on

BEGIN;
SELECT 1;
SELECT 1/0;
SELECT 2;
COMMIT;
```

Фрагменты вывода:

Задача 1:

```
CREATE TABLE

INSERT 0 1

 ctid  | xmin | xmax | id
-------+------+------+----
 (0,1) |  815 |    0 |  1
(1 row)


UPDATE 1

 ctid  | xmin | xmax | id
-------+------+------+----
```

```
 (0,2) |  816 |    0 |  2
(1 row)


UPDATE 1

 ctid  | xmin | xmax | id
-------+------+------+----
 (0,3) |  817 |    0 |  3
(1 row)


DELETE 1

 ctid | xmin | xmax | id
------+------+------+----
(0 rows)


CREATE EXTENSION

 lp | lp_off | lp_flags | lp_len | t_xmin | t_xmax | t_field3 | t_ctid |
t_infomask2 | t_infomask | t_hoff | t_bits | t_oid |   t_data
----+--------+----------+--------+--------+--------+----------+--------+----------
---+------------+--------+--------+-------+------------
  1 |   8160 |        1 |     28 |    815 |    816 |        0 | (0,2)  |
16385 |       1280 |     24 |        |       | \x01000000
  2 |   8128 |        1 |     28 |    816 |    817 |        0 | (0,3)  |
49153 |       9472 |     24 |        |       | \x02000000
  3 |   8096 |        1 |     28 |    817 |    818 |        0 | (0,3)  |
40961 |       9472 |     24 |        |       | \x03000000
(3 rows)
```

```
 t_xmin | t_xmax | t_ctid
--------+--------+--------
    836 |    837 | (0,2)
    837 |    838 | (0,3)
    838 |    839 | (0,3)
(3 rows)
```

Задача 2:

```
  ctid
--------
 (7,65)
(1 row)
```

```
 lp | t_xmin | t_xmax | t_ctid
----+--------+--------+--------
  2 |    775 |    776 | (0,2)
  3 |    798 |    799 | (0,3)
  4 |    800 |      0 | (0,4)
 46 |    270 |      0 | (0,46)
 47 |    522 |      0 | (0,47)
(5 rows)
```

Задача 3:

```
BEGIN

 ?column?
----------
        1
(1 row)


ERROR:  division by zero

 ?column?
----------
        2
(1 row)

COMMIT
```

Часть 4. Снимки данных (Snapshots)

Выполненные задачи:

1. Воспроизвёл ситуацию, при которой одна транзакция (A) видит строку, а другая (B), начавшаяся позже, — уже нет (строка удалена и зафиксирована после начала A, но до начала B). Использовал функции `pg_current_snapshot()` и `pg_snapshot_xip(pg_current_snapshot())` для анализа снимков обеих транзакций. Изучил значения `xmin` и `xmax` удаленной строки. Снимок транзакции A — 879:879:, t_xmin = 878, вставка видна. t_xmax = 880 > xmax = 879. Снимок транзакции B — 881:881:, t_xmax = 880 < xmin = 881 и удаляющая транзакция к этому моменту зафиксирована - строка невидима.

2. Создал функцию `STABLE`, возвращающую данные из таблицы. Исследовал, какой снимок данных используется для запроса внутри этой функции при разных уровнях изоляции (`Read Committed` и `Repeatable Read`). Повторил для функции `VOLATILE`. На `Read Committed` обе функции вернули новое значение, так как запрос внутри функции использует снимок текущего оператора, а снимок на `Read Committed` меняется от запроса к запросу. На `Repeatable Read` обе функции неизменно

возвращают одно и то же — они читают снимок транзакции, зафиксированный первым запросом. `STABLE` и `VOLATILE` не влияет на видимость данных (снимок), только на то, может ли планировщик переиспользовать результат внутри одного оператора.

3. В транзакции 1 (уровень `Repeatable Read`) экспортировал снимок данных с помощью `pg_export_snapshot()`. В транзакции 2 изменил данные и зафиксировал. В транзакции 3 импортировал снимок из транзакции 1 (`SET TRANSACTION SNAPSHOT '...';`). Убедился, что в транзакции 3 видны данные в состоянии на момент экспорта снимка, до изменений из транзакции 2.

Команды:

Задача 1:

```
-- Транзакция A (Сеанс 1)
CREATE TABLE snapshot_test (id INT, data TEXT);
INSERT INTO snapshot_test VALUES (1, 'first row');
COMMIT;

BEGIN ISOLATION LEVEL REPEATABLE READ;
SELECT * FROM snapshot_test;

SELECT pg_current_snapshot();
SELECT pg_snapshot_xip(pg_current_snapshot());
```

```
-- Сеанс 2
DELETE FROM snapshot_test WHERE id = 1;
COMMIT;
```

```
-- Транзакция B (Сеанс 3)
BEGIN ISOLATION LEVEL REPEATABLE READ;
SELECT * FROM snapshot_test;

SELECT pg_current_snapshot();
SELECT pg_snapshot_xip(pg_current_snapshot());
```

```
-- Транзакция A (Сеанс 1)
SELECT * FROM snapshot_test;
SELECT xmin, xmax, cmin, cmax, ctid, * FROM snapshot_test;
COMMIT;
```

Задача 2:

```
-- Создание таблицы
CREATE TABLE func_test (id INT, data TEXT);
INSERT INTO func_test VALUES (1, 'initial');
```

```
-- Функция STABLE
CREATE OR REPLACE FUNCTION get_data_stable()
RETURNS TABLE(id INT, data TEXT) AS $$
  SELECT * FROM func_test;
$$ LANGUAGE SQL STABLE;

-- Функция VOLATILE
CREATE OR REPLACE FUNCTION get_data_volatile()
RETURNS TABLE(id INT, data TEXT) AS $$
  SELECT * FROM func_test;
$$ LANGUAGE SQL VOLATILE;
```

```
-- Сеанс 1
BEGIN;
SELECT * FROM func_test;

SELECT * FROM get_data_stable();

SELECT * FROM get_data_volatile();
```

```
-- Сеанс 2
INSERT INTO func_test VALUES (2, 'added in session 2');
COMMIT;
```

```
-- Сеанс 1
SELECT * FROM func_test;

SELECT * FROM get_data_stable();

SELECT * FROM get_data_volatile();

COMMIT;
```

```
-- Сеанс 1
BEGIN ISOLATION LEVEL REPEATABLE READ;
```

```
SELECT * FROM func_test;

SELECT pg_current_snapshot();
```

```
-- Сеанс 2
INSERT INTO func_test VALUES (3, 'added in RR test');
COMMIT;
```

```
-- Сеанс 1
SELECT * FROM func_test;

SELECT * FROM get_data_stable();

SELECT * FROM get_data_volatile();

SELECT * FROM get_data_volatile();

COMMIT;
```

Задача 3:

```
-- Сеанс 1 (экспорт снимка)
DROP TABLE IF EXISTS export_test;
CREATE TABLE export_test (id INT, data TEXT);
INSERT INTO export_test VALUES (1, 'original data');
COMMIT;

BEGIN ISOLATION LEVEL REPEATABLE READ;

SELECT * FROM export_test;

SELECT pg_export_snapshot();
```

```
-- Сеанс 2 (изменение данных)
INSERT INTO export_test VALUES (2, 'new data');
UPDATE export_test SET data = 'modified' WHERE id = 1;
COMMIT;

SELECT * FROM export_test;
```

```

```

```
-- Сеанс 3 (импорт снимка)
BEGIN ISOLATION LEVEL REPEATABLE READ;
SET TRANSACTION SNAPSHOT '00000003-0000008A-1';

SELECT * FROM export_test;
SELECT xmin, xmax, ctid, * FROM export_test;

COMMIT;
```

```
-- Сеанс 1
SELECT * FROM export_test;
COMMIT;

SELECT * FROM export_test;
```

Фрагменты вывода:

Задача 1

```
CREATE TABLE

INSERT 0 1
```

```
CREATE FUNCTION

CREATE FUNCTION
```

```
-- Сеанс 1
BEGIN

 id |  data
----+---------
  1 | initial
(1 row)

 id |  data
----+---------
  1 | initial
(1 row)

 id |  data
----+---------
```

```
  1 | initial
(1 row)
```

```
-- Сеанс 2
INSERT 0 1

WARNING:  there is no transaction in progress
COMMIT
```

```
-- Сеанс 1

 id |        data
----+--------------------
  1 | initial
  2 | added in session 2
(2 rows)

 id |        data
----+--------------------
  1 | initial
  2 | added in session 2
(2 rows)

 id |        data
----+--------------------
  1 | initial
  2 | added in session 2
(2 rows)

COMMIT
```

Задача 2:

```
CREATE FUNCTION

CREATE FUNCTION

CREATE FUNCTION
```

```
Тест 1: READ COMMITTED
-- Сеанс 1
BEGIN
 f_stable | f_volatile | pg_current_snapshot
----------+------------+---------------------
```

```
        1 |          1 | 910:910:
(1 row)
```

```
-- Сеанс 2
INSERT 0 1

WARNING:  there is no transaction in progress
COMMIT
```

```
-- Сеанс 1
 f_stable | f_volatile | pg_current_snapshot
----------+------------+---------------------
        2 |          2 | 911:911:
(1 row)

COMMIT
```

```
-- Сеанс 1
BEGIN
 f_stable | f_volatile | pg_current_snapshot
----------+------------+---------------------
        2 |          2 | 911:911:
(1 row)
```

```
-- Сеанс 2
INSERT 0 1

WARNING:  there is no transaction in progress
COMMIT

 f_stable | f_volatile | pg_current_snapshot
----------+------------+---------------------
        3 |          3 | 912:912:
```

```
-- Сеанс 1
 f_stable | f_volatile | pg_current_snapshot
----------+------------+---------------------
        2 |          2 | 911:911:
(1 row)

COMMIT
```

```
Тест 2: REPEATABLE READ
-- Сеанс 1
BEGIN

 id |  data
----+---------
  1 | initial
(1 row)

 pg_current_snapshot
---------------------
 834:834:
(1 row)
```

```
-- Сеанс 2
INSERT 0 1

WARNING:  there is no transaction in progress
COMMIT
```

```
-- Сеанс 1

 id |  data
----+---------
  1 | initial
(1 row)

 id |  data
----+---------
  1 | initial
(1 row)

 id |  data
----+---------
  1 | initial
(1 row)

 id |  data
----+---------
  1 | initial
(1 row)

COMMIT
```

Задача 3:

```
-- Сеанс 1 (экспорт снимка)
CREATE TABLE

INSERT 0 1

WARNING:  there is no transaction in progress
COMMIT

BEGIN

 id |     data
----+---------------
  1 | original data
(1 row)

 pg_export_snapshot
---------------------
 00000003-0000008A-1
(1 row)


-- Сеанс 2 (изменение данных)
INSERT 0 1

UPDATE 1

WARNING:  there is no transaction in progress
COMMIT

 id |   data
----+----------
  2 | new data
  1 | modified
(2 rows)


-- Сеанс 3 (импорт снимка)

BEGIN

SET

 id |     data
----+---------------
  1 | original data
(1 row)

 xmin | xmax | ctid  | id |     data
------+------+-------+----+---------------
  836 |  838 | (0,1) |  1 | original data
(1 row)

COMMIT
```

```
-- Сеанс 1

 id |     data
----+---------------
  1 | original data
(1 row)

COMMIT

 id |   data
----+----------
  2 | new data
  1 | modified
(2 rows)
```

Результаты выполнения

1. **Сравнение уровней изоляции READ COMMITTED и REPEATABLE READ.**
   Практические эксперименты продемонстрировали принципиальные различия в формировании снимков данных:

   - При `READ COMMITTED` каждая SQL-команда получает актуальный снимок, поэтому строки, удалённые другими зафиксированными транзакциями, немедленно исчезают из результатов повторных запросов.
   - При `REPEATABLE READ` снимок фиксируется в момент первого обращения к данным, обеспечивая стабильное представление на протяжении всей транзакции — удалённые извне строки остаются видимыми до завершения транзакции.

   Это подтверждает, что PostgreSQL реализует неблокирующее согласованное чтение через механизм версионности.

2. **Транзакционность DDL-операций и изоляция объектов схемы.**
   Эксперименты с созданием таблиц внутри транзакций показали:

   - Новые объекты невидимы другим сеансам до выполнения `COMMIT`.
   - Операция `ROLLBACK` полностью отменяет создание объектов.
   - DDL-команды участвуют в MVCC наравне с DML, что отличает PostgreSQL от многих других СУБД.

   Дополнительно продемонстрирована блокировка `DROP TABLE` при наличии активных `SELECT`-запросов в других транзакциях (конфликт блокировок ACCESS SHARE и ACCESS EXCLUSIVE).

3. **Выявление аномалий параллелизма: фантомное чтение.**
   На уровне `READ COMMITTED` зафиксировано классическое фантомное чтение: строки, вставленные параллельной транзакцией после первого запроса, появились в результатах повторного запроса той же команды. При уровне `REPEATABLE READ` такие вставки остаются невидимыми, что подтверждает защиту от неповторяемого чтения, хотя фантомы всё ещё теоретически возможны в определённых сценариях (предотвращаются только на уровне SERIALIZABLE).

4. **Исследование версий строк через расширение pageinspect.**
   Последовательное выполнение `INSERT`, двух `UPDATE` и `DELETE` привело к созданию четырёх физических версий одной логической строки:

   - Исходная версия (`t_xmin` = вставившая транзакция, `t_xmax` = первый UPDATE).
   - Две промежуточные версии от обновлений, каждая со своими `t_xmin`/`t_xmax`.
   - Финальная удалённая версия (`t_xmax` ≠ 0, `t_ctid` указывает на саму себя).

   Анализ через `heap_page_items()` наглядно показал цепочку версий и механизм отслеживания устаревших данных до выполнения `VACUUM`.

5. **Анализ системных полей и работа со страницами системных каталогов.**
   При помощи `pageinspect` определено физическое расположение строки `pg_class`, описывающей саму себя. Подсчёт видимых версий (`lp_flags = 1`) в странице системной таблицы продемонстрировал применение MVCC даже к метаданным PostgreSQL, что обеспечивает согласованность запросов к системным каталогам.

6. **Режим автоматических точек сохранения ON_ERROR_ROLLBACK.**
   Включение параметра `\set ON_ERROR_ROLLBACK on` в psql позволило продолжить работу транзакции после возникновения ошибки (деление на ноль). PostgreSQL автоматически создал `SAVEPOINT` перед каждой командой, откатив только проблемную операцию без прерывания всей транзакции. Это упрощает интерактивную отладку сложных скриптов.

7. **Анализ снимков транзакций и видимость версий строк.**
   Используя функции `pg_current_snapshot()` и `pg_snapshot_xip()`, проанализированы снимки двух параллельных транзакций (A и B):

   - Транзакция A (более ранняя) видит строку, так как удаляющая транзакция началась после создания её снимка (`xmax` > `xmin` снимка A).
   - Транзакция B (более поздняя) не видит строку, поскольку удаляющая транзакция завершилась до создания снимка B.

   Это демонстрирует, как значения `xmin`/`xmax` строки и границы снимка транзакции определяют видимость данных в MVCC.

8. **Поведение функций STABLE и VOLATILE на разных уровнях изоляции.**
   Эксперименты показали:

   - В `READ COMMITTED` обе категории функций видят изменения между вызовами, так как каждая команда формирует новый снимок.
   - В `REPEATABLE READ` даже функции `VOLATILE` используют фиксированный снимок транзакции, что отличается от поведения в некоторых других СУБД.

   Атрибуты `STABLE`/`VOLATILE` влияют в первую очередь на оптимизацию запросов (кэширование результатов), а не на семантику видимости в PostgreSQL.

9. **Экспорт и импорт снимков между транзакциями.**
   С помощью `pg_export_snapshot()` создан идентификатор снимка в транзакции 1, который затем был импортирован в транзакцию 3 через `SET TRANSACTION SNAPSHOT`. Это позволило транзакции 3 видеть данные в точности в том состоянии, которое существовало на момент экспорта,

игнорируя все изменения транзакции 2. Механизм применим для создания согласованных резервных копий и распределённых аналитических запросов, требующих единого представления данных.

10. **Общие выводы о механизме MVCC в PostgreSQL.**
    Лабораторная работа подтвердила, что MVCC обеспечивает:

    - Высокий уровень параллелизма без блокировок на чтение.
    - Изоляцию транзакций через версионность данных и управление снимками.
    - Транзакционность не только DML, но и DDL-операций.
    - Прозрачное управление устаревшими версиями строк до момента очистки.

    Использование системных полей (`xmin`, `xmax`, `ctid`) и расширения `pageinspect` позволяет глубоко анализировать внутреннее устройство таблиц и принципы работы версионности на физическом уровне.

Выводы

1. Изучил принципы многоверсионного управления конкурентным доступом (MVCC) в PostgreSQL.
2. Получил практические навыки наблюдения за работой MVCC, анализа версий строк, снимков данных и уровней изоляции транзакций.
3. Освоил использование расширений и системных представлений для исследования внутренней структуры данных.