МИНОБРНАУКИ РОССИИ

Федеральное государственное бюджетное образовательное учреждение

высшего образования

«МИРЭА – Российский технологический университет»

РТУ МИРЭА

Институт комплексной безопасности и специального приборостроения

Кафедра КБ-14 «Интеллектуальные системы информационной безопасности»

Политики безопасности баз данных

Практическая работа № 2

Работа со сторонними базами данными. Построение и оптимизация

ОТЧЁТ

Выполнил студент группы
БСБО-07-20
Любовский С.В.

Москва 2023

Выполнение задания 1.

1. Создайте новую базу данных в PostgreSQL включающие две таблицы: "accounts" и "transactions". Таблица "accounts" должна содержать следующие поля: id (уникальный идентификатор), name (имя), balance (баланс). Таблица "transactions" должна содержать следующие поля: id (уникальный идентификатор), account_id (ссылка на id в таблице "accounts"), amount (сумма).

```
-- Создадим таблицы аккаунтов и транзацкий
CREATE TABLE accounts (
    id serial PRIMARY KEY,
    name varchar(255) NOT NULL,
    balance money NOT NULL
);

CREATE TABLE transactions (
    id serial PRIMARY KEY,
    account_id integer NOT NULL REFERENCES accounts(id),
    amount money NOT NULL
);
```

2. Проведите проверку что PostgreSQL не допускается аномалия **грязного чтения**, объясните почему.

```
-- Создадим временную роль для тестирования
CREATE ROLE test_role;
GRANT ALL ON ALL TABLES IN SCHEMA "public" TO test_role;
GRANT ALL ON ALL SEQUENCES IN SCHEMA "public" TO test_role;

SET ROLE test_role;

-- Создадим незавершенную транзакцию
SELECT setval('accounts_id_seq', 1, false);

BEGIN;
INSERT INTO accounts (name, balance) VALUES ('test', 1000);
INSERT INTO transactions (account_id, amount) VALUES (1, 100);

-- Проверим, что транзакция в процессе (результат должен быть не пустым)
SELECT txid_current_if_assigned();

-- В новом окне подключимся к БД и выполним запрос из под пользователя
postgres
BEGIN;
SELECT * FROM accounts;
SELECT * FROM transactions;
COMMIT;
```

Данные запросы не видят данные, созданные в другой транзакции, которая не была закончена. Это происходит, потому что PostgreSQL, как и многие другие СУБД, использует механизмы изоляции транзакций для предотвращения аномалий, включая грязное чтение.

3. Проверьте, что на уровне изоляции Read Committed не предотвращается аномалия фантомного чтения.

Проверим, что на уровне изоляции Read Committed не предотвращается аномалия фантомного чтения. Для этого создадим незавершенную транзакцию на уровне Read Committed, которая будет выводить данные таблицы «accounts».

```
-- Создадим транзакцию для выборки данных из таблицы accounts с уровнем
изоляции READ COMMITTED
BEGIN TRANSACTION ISOLATION LEVEL READ COMMITTED;
SELECT * FROM accounts;

-- В другой сессии изменяем данные в таблице accounts
BEGIN TRANSACTION ISOLATION LEVEL READ COMMITTED;
INSERT INTO accounts (id, name, balance) VALUES (4, 'test', 1000);
COMMIT;

-- В первой сессии видим изменения
SELECT * FROM accounts;
```

Сессия 1:

```
practice2=# BEGIN TRANSACTION ISOLATION LEVEL READ COMMITTED;
BEGIN
practice2=*# SELECT * FROM accounts;
 id | name | balance
----+------+---------
(0 rows)

practice2=*# SELECT * FROM accounts;
 id | name |  balance
----+------+-----------
  4 | test | $1,000.00
(1 row)

practice2=*#
```

Сессия 2:

```
postgres=# \c practice2
You are now connected to database "practice2" as user "postgres".
practice2=# BEGIN TRANSACTION ISOLATION LEVEL READ COMMITTED;
BEGIN
practice2=*# INSERT INTO accounts (id, name, balance) VALUES (4, 'test', 1000);
INSERT 0 1
practice2=*# COMMIT;
COMMIT
practice2=# _
```

4. Начните транзакцию с уровнем изоляции Repeatable Read (и пока не выполняйте в ней никаких команд). В другом сеансе удалите строку и зафиксируйте изменения. Видна ли строка в открытой транзакции? Что изменится, если в начале транзакции выполнить запрос, но не обращаться в нем ни к одной таблице?

Создадим две транзакции – одна будет пустой и иметь уровень изоляции repeatable read, вторая удалит строчку

```
--- Добавим запись в таблицу accounts
INSERT INTO accounts (id, name, balance) VALUES (100, 'test', 1000);

--- Откроем транзакцию с уровнем REPETABLE READ
BEGIN TRANSACTION ISOLATION LEVEL REPEATABLE READ;

--- В другой сессии удалим строку и таблицы accounts
BEGIN TRANSACTION ISOLATION LEVEL REPEATABLE READ;
DELETE FROM accounts WHERE id = 100;
COMMIT;

--- В первой сессии выведем таблицу accounts
SELECT * FROM accounts;
```

Сессия 1:

```
practice2=# BEGIN TRANSACTION ISOLATION LEVEL REPEATABLE READ;
BEGIN
practice2=*# SELECT * FROM accounts;
 id | name | balance
----+------+---------
(0 rows)

practice2=*# _
```

Сессия 2:

```
practice2=# BEGIN TRANSACTION ISOLATION LEVEL REPEATABLE READ;
BEGIN
practice2=*# DELETE FROM accounts WHERE id = 100;
DELETE 1
practice2=*# COMMIT;
COMMIT
practice2=# _
```

После обращения к таблице из первой транзакции мы видим, что данных в таблице нет.

Теперь добавим в первую транзакцию запрос, не затрагивающий ни одну другую таблицу.

```sql
--- Снова добавим запись в таблицу accounts
INSERT INTO accounts (id, name, balance) VALUES (100, 'test', 1000);

--- Откроем транзакцию с уровнем REPETABLE READ и сделаем SELECT который не
затрагивает ни одну из таблиц
BEGIN TRANSACTION ISOLATION LEVEL REPEATABLE READ;
SELECT 1;

--- В другой сессии удалим строку и таблицы accounts
BEGIN TRANSACTION ISOLATION LEVEL REPEATABLE READ;
DELETE FROM accounts WHERE id = 100;
COMMIT;

--- В первой сессии выведем таблицу accounts
SELECT * FROM accounts;
```

Сессия 1:

```
practice2=# BEGIN TRANSACTION ISOLATION LEVEL REPEATABLE READ;
BEGIN
practice2=*# SELECT 1;
 ?column?
----------
        1
(1 row)

practice2=*# SELECT * FROM accounts;
 id  | name |  balance
-----+------+-----------
 100 | test | $1,000.00
(1 row)

practice2=*# _
```

Сессия 2:

```
practice2=# BEGIN TRANSACTION ISOLATION LEVEL REPEATABLE READ;
BEGIN
practice2=*# DELETE FROM accounts WHERE id = 100;
DELETE 1
practice2=*# COMMIT;
COMMIT
practice2=# _
```

В этот раз данные остались видны в первой транзакции.

5. Напишите функцию, которая позволяет выполнить перевод средств с одного счета на другой, используя транзакции. Функция должна использовать уровень изоляции транзакции "Serializable". Протестируйте функцию с использованием нескольких параллельных сеансов, чтобы убедиться, что переводы не могут быть выполнены дважды.

```sql
--- Создадим функции для перевода денег с одного счета на другой
CREATE OR REPLACE FUNCTION transfer_money(
    p_from_acc int,
    p_to_acc int,
    p_amount money
)
RETURNS void AS $$
BEGIN
    UPDATE
        accounts
    SET
        balance = balance - p_amount
    WHERE
        id = p_from_acc;

    UPDATE
        accounts
    SET
        balance = balance + p_amount
    WHERE
        id = p_to_acc;

    INSERT INTO
        transactions (account_id, amount)
    VALUES
        (p_from_acc, -1 * p_amount),
        (p_to_acc, p_amount);
END;
$$ LANGUAGE plpgsql;

--- Создадим два аккаунта
INSERT INTO
    accounts (id, name, balance)
VALUES
    (10, 'Alice', 100),
```

```
    (11, 'Bob', 50);

--- Переведем 10 долларов с аккаунта Алисы на аккаунт Боба двумя
паралельными транзакиями с уровнем изоляции SERIALIZABLE.
BEGIN TRANSACTION ISOLATION LEVEL SERIALIZABLE;
SELECT transfer_money(10, 11, money(80)); -- 80 + 80 > 100

--- Попытаемся применить изменения в обеих транзакциях
COMMIT;
```

Сессия 1:

```
practice2=# BEGIN TRANSACTION ISOLATION LEVEL SERIALIZABLE;
BEGIN
practice2=*# SELECT transfer_money(10, 11, money(80)); -- 80 + 80 > 100
 transfer_money
----------------

(1 row)

practice2=*# commit
practice2-*# ;
COMMIT
practice2=#
```

Сессия 2:

```
practice2=# BEGIN TRANSACTION ISOLATION LEVEL SERIALIZABLE;
BEGIN
practice2=*# SELECT transfer_money(10, 11, money(80)); -- 80 + 80 > 100
ERROR:  could not serialize access due to concurrent update
CONTEXT:  SQL statement "UPDATE
        accounts
    SET
        balance = balance - p_amount
    WHERE
        id = p_from_acc"
PL/pgSQL function transfer_money(integer,integer,money) line 3 at SQL st
atement
practice2=!# _
```

Мы видим, что при выполнении функции в первой транзакции вторая блокируется. После применения первой транзакции вторая упадет с ошибкой и не применится.

6. Начните транзакцию Repeatable Read и выполните какой-нибудь запрос. В другом сеансе создайте таблицу. Видно ли в первой транзакции описание таблицы в системном каталоге? Можно ли в ней прочитать строки таблицы?

Создадим транзакцию с уровнем изоляции Repeatable Read, и выполним какой-нибудь запрос. Также, в другом сеансе, создадим новую таблицу. В

итоге, в первой транзакции описание таблицы в системной каталоге видно. Прочитать строки в это таблицы возможность отсутствует.

7. Убедитесь, что команда DROP TABLE транзакционна.

```
--- Дропнем таблицу в транзакции
BEGIN;
DROP TABLE accounts CASCADE;

--- Убедимся, что таблица была удалена
SELECT table_name FROM information_schema.tables
WHERE table_schema NOT IN ('information_schema', 'pg_catalog')
AND table_schema IN('public', 'myschema');

--- Откатим транзакцию
ROLLBACK;

--- Убедимся, что таблица не была удалена
SELECT * FROM accounts;
```

Сессия:

```
practice2=# BEGIN;
BEGIN
practice2=*# DROP TABLE accounts CASCADE;
NOTICE:  drop cascades to constraint transactions_account_id_fkey on tab
le transactions
DROP TABLE
practice2=*# ROLLBACK;
ROLLBACK
practice2=# SELECT * FROM accounts;
 id | name  | balance
----+-------+---------
 10 | Alice |  $20.00
 11 | Bob   | $130.00
(2 rows)

practice2=# _
```

Операция DROP (CASCADE) является транзакционной.

Выполнение задания 2.

1. Установите расширение pageinspect.
2. Создать базу данных с именем versions_db. Создать таблицу users со следующими полями:
    a. id: уникальный идентификатор пользователя (integer, primary key, auto-increment).
    b. username: имя пользователя (varchar(255)).
    c. email: электронный адрес пользователя (varchar(255)).
    d. version: версия строки (integer).

```
--- Создадим базу данных versions_db
CREATE DATABASE versions_db;

--- Создадим таблицу users
CREATE TABLE users (
    id serial PRIMARY KEY,
    username varchar(255) NOT NULL,
    email varchar(255) NOT NULL,
    version integer NOT NULL DEFAULT 1
)
```

3. Создать триггер, который будет автоматически увеличивать версию строки при любом обновлении.

```
--- Создадим триггер, который будет автоматически увеличивать версию строки
при любом обновлении
CREATE OR REPLACE FUNCTION update_version() RETURNS TRIGGER AS
$$
BEGIN
    NEW.version = OLD.version + 1;
    RETURN NEW;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER update_version_trigger
    BEFORE UPDATE ON users
    FOR EACH ROW
    EXECUTE FUNCTION update_version();
```

4. Вставить в таблицу users строку с различными данными, а затем обновите.

```
versions_db=# INSERT INTO users (username, email) VALUES ('user1', 'user
1@temp.ru');
INSERT INTO users (username, email) VALUES ('user2', 'user2@temp.ru');

UPDATE users SET email = 'user1@temp1.ru' WHERE username = 'user1';

SELECT * FROM users;
INSERT 0 1
INSERT 0 1
UPDATE 1
 id | username |     email      | version
----+----------+----------------+---------
  4 | user2    | user2@temp.ru  |       1
  3 | user1    | user1@temp1.ru |       2
(2 rows)

versions_db=# _
```

5. При помощи следующего запроса:

```
SELECT '(0,'||lp||')' AS ctid,
    t_xmin as xmin,
    t_xmax as xmax,
    CASE WHEN (t_infomask & 256) > 0  THEN 't' END AS xmin_c,
    CASE WHEN (t_infomask & 512) > 0  THEN 't' END AS xmin_a,
    CASE WHEN (t_infomask & 1024) > 0 THEN 't' END AS xmax_c,
    CASE WHEN (t_infomask & 2048) > 0 THEN 't' END AS xmax_a
FROM heap_page_items(get_raw_page('users',0))
ORDER BY lp;
```

Где,

- **ctid** является ссылкой на следующую, более новую, версию той же строки. У самой новой, актуальной, версии строки ctid ссылается на саму эту версию
- **xmin** и **xmax** определяют видимость данной версии строки в терминах начального и конечного номеров транзакций.
- **xmin_c, xmin_a, xmax_c, xmax_a** содержит ряд битов, определяющих свойства данной версии

Выведите информацию о версиях строк, узнав сколько версий строк щас находится в таблице и сравнить их с атрибутом (version)

```
versions_db=# SELECT '(0,'||lp||')' AS ctid,
       t_xmin as xmin,
       t_xmax as xmax,
       CASE WHEN (t_infomask & 256) > 0  THEN 't' END AS xmin_c,
       CASE WHEN (t_infomask & 512) > 0  THEN 't' END AS xmin_a,
       CASE WHEN (t_infomask & 1024) > 0 THEN 't' END AS xmax_c,
       CASE WHEN (t_infomask & 2048) > 0 THEN 't' END AS xmax_a
FROM heap_page_items(get_raw_page('users',0))
ORDER BY lp;
 ctid  | xmin | xmax | xmin_c | xmin_a | xmax_c | xmax_a
-------+------+------+--------+--------+--------+--------
 (0,1) |  767 |  769 | t      |        | t      |
 (0,2) |  768 |    0 | t      |        |        | t
 (0,3) |  769 |  769 | t      |        |        |
(3 rows)
```

6. Опустошим таблицу при помощи **TRUNCATE;**
7. Начините транзакцию и вставьте новую строку и узнайте номер текущей транзакции (это можно сделать при помощи след команды: **INSERT INTO users(...) VALUES (...) RETURNING *, ctid, xmin, xmax;**

```
versions_db=# BEGIN;
INSERT INTO users(username, email) VALUES ('user1', 'user1@tmp.ru') RETU
RNING *, ctid, xmin, xmax;
BEGIN
 id | username |    email     | version | ctid  | xmin | xmax
----+----------+--------------+---------+-------+------+------
  7 | user1    | user1@tmp.ru |       1 | (0,1) |  771 |    0
(1 row)

INSERT 0 1
versions_db=*#
```

8. Поставьте точку сохранения и добавьте новую строку использовав команду из пункта 7.

```
versions_db=*# SAVEPOINT savepoint1;
INSERT INTO users(username, email) VALUES ('user2', 'user2@tmp.ru') RETU
RNING *, ctid, xmin, xmax;
SAVEPOINT
 id | username |    email     | version | ctid  | xmin | xmax
----+----------+--------------+---------+-------+------+------
  8 | user2    | user2@tmp.ru |       1 | (0,2) |  772 |    0
(1 row)

INSERT 0 1
versions_db=*# _
```

9. Откатимся к точке сохранения и добавим новую строчку аналогично 7 и 8 пункту.

```
versions_db=*# ROLLBACK TO savepoint1;
INSERT INTO users(username, email) VALUES ('user3', 'user3@tmp.ru') RETU
RNING *, ctid, xmin, xmax;
ROLLBACK
 id | username |    email     | version | ctid  | xmin | xmax
----+----------+--------------+---------+-------+------+------
  9 | user3    | user3@tmp.ru |       1 | (0,3) |  773 |    0
(1 row)

INSERT 0 1
```

10. Выведем сведения о версиях строк.

```
versions_db=*# SELECT '(0,'||lp||')' AS ctid,
    t_xmin as xmin,
    t_xmax as xmax,
    CASE WHEN (t_infomask & 256) > 0  THEN 't' END AS xmin_c,
    CASE WHEN (t_infomask & 512) > 0  THEN 't' END AS xmin_a,
    CASE WHEN (t_infomask & 1024) > 0 THEN 't' END AS xmax_c,
    CASE WHEN (t_infomask & 2048) > 0 THEN 't' END AS xmax_a
FROM heap_page_items(get_raw_page('users',0))
ORDER BY lp;

 ctid  | xmin | xmax | xmin_c | xmin_a | xmax_c | xmax_a
-------+------+------+--------+--------+--------+--------
 (0,1) |  771 |    0 |        |        |        | t
 (0,2) |  772 |    0 |        |        |        | t
 (0,3) |  773 |    0 |        |        |        | t
(3 rows)

versions_db=*# _
```

Выполнение задания 3.

1. Создать таблицу t с полями id(integer) и name (char(2000)) с параметром filfactor = 75%.

```
--- Создадим таблицу «t» с полями: id, name (с параметром filfactor = 75%).
CREATE TABLE t (
    id INT,
    name VARCHAR(2000)
) WITH (FILFACTOR = 75);
```

2. Создать индекс над полем t(name)

```
--- Создадим индекс над полем t(name).
CREATE INDEX t_name_idx ON t (name);
```

3. Установить расширение *pageinspect.*

4. Создать представление, которое будет включать в себя информацию о версиях строк при помощи след запроса:

```
CREATE VIEW t_v AS
SELECT '(0,'||lp||')' AS ctid,
    CASE lp_flags
      WHEN 0 THEN 'unused'
      WHEN 1 THEN 'normal'
      WHEN 2 THEN 'redirect to '||lp_off
      WHEN 3 THEN 'dead'
    END AS state,
    t_xmin || CASE
      WHEN (t_infomask & 256) > 0 THEN ' (c)'
      WHEN (t_infomask & 512) > 0 THEN ' (a)'
      ELSE ''
    END AS xmin,
    t_xmax || CASE
      WHEN (t_infomask & 1024) > 0 THEN ' (c)'
      WHEN (t_infomask & 2048) > 0 THEN ' (a)'
      ELSE ''
    END AS xmax,
    CASE WHEN (t_infomask2 & 16384) > 0 THEN 't' END AS
hhu,
    CASE WHEN (t_infomask2 & 32768) > 0 THEN 't' END AS
hot,
    t_ctid
FROM heap_page_items(get_raw_page('t',0))
ORDER BY lp;
```

- флаг Heap Hot Updated показывает, что надо идти по цепочке ctid,
- флаг Heap Only Tuple показывает, что на данную версию строки нет ссылок из индексов.

5. Спроецировать ситуацию в таблице t, при которой произойдет внутристраничная очистка без участия HOT-обновлений.

```
--- Спроецируем ситуацию в таблице «t», при которой произойдет внутри
страничная очистка без участия HOT-обновлений
INSERT INTO t (id, name) VALUES (1,
'1111111111111111111111111111111111111111111111111111111111111111
1111111111111111111111111111111111111111);
```

6. После воспроизвести ситуацию, но уже с HOT-обновлением

```
--- После воспроизведем ситуацию, но уже с HOT-обновлением.
```

```
UPDATE t SET id=2 WHERE id=1;

--- Посмотрим, что получилось.
SELECT * FROM t_v;
```

```
versions_db=# UPDATE t SET id=2 WHERE id=1;
UPDATE 1
versions_db=# SELECT * FROM t_v;
 ctid  | state  |  xmin   | xmax  | hhu | hot | t_ctid
-------+--------+---------+-------+-----+-----+--------
 (0,1) | normal | 777 (c) | 778   | t   |     | (0,2)
 (0,2) | normal | 778     | 0 (a) |     | t   | (0,2)
(2 rows)
```

Звездочка.

Использовались факторы заполнения меньше стандартного значения выгодно только при большом количестве вставок в таблицу, однако при этом размер индекса будет расти значительно быстрее чем количество данных. При использовании фактора заполнения больше стандартного скорость работы индекса увеличивается, его размер уменьшается, но вставки и обновления будут занимать больше времени.