



Дисципліна «Операційні системи»

Лабораторна робота №10



INFORMATION
SYSTEMS

department

Тема: «Керування процесами-транзакціями в базах даних»

Викладач: Олександр А. Блажко,

доцент кафедри ІС Одеської політехніки, blazhko@ieee.org

Мета роботи: дослідити поведінку процесів-транзакцій в базах даних та засоби керування ними через механізм блокування з використанням сучасних систем керування базами даних на прикладі *PostgreSQL*.

План.

1 Теоретичні відомості

2 Завдання

1 Теоретичні відомості

1.1 Основи керування транзакціями-процесами

1.1.1 Основи транзакційної роботи процесів

Процеси, які виконуються одночасно, завжди конкурують за ресурси.

Основним ресурсом конкурування є дані.

Дані можуть мати різний рівень деталізації та розміщення:

- змінна в оперативній пам'яті, наприклад, *int B*;
- масив змінних в оперативній пам'яті, наприклад, *int M1={1,2}*;
- структура даних з декількома змінними в оперативній пам'яті, наприклад,

Struct S1 (int B; int D);

- рядок символів у файлі файлової системи;
- цілий файл файлової системи.

В сучасних інформаційних системах процеси, які працюють з даними, часто називають *Транзакціями*.

Транзакція (англ. *Transactions* – ділова угода між учасниками) – це:

- впорядкована послідовність *CRUD*-операцій з даними (*Create A, Read B, Update B, Delete D*);
- оперативна зміна значень даних в оперативній пам'яті при виконанні операцій до моменту завершення транзакції;
- перенос всіх змінених даних з оперативної пам'яті у файл на диску під час завершення транзакції;

- задоволення *ACID*-вимог роботи з даними.

ACID-вимоги – (*Atomicity-Consistency-Isolation-Durability*):

- *Атомарність (Atomicity)* - виконується або вся транзакція цілком з усіма її *CRUD*-операціями, або всі її результати операцій скасовуються;
- *Узгодженість (Consistency)* - транзакція переводить дані із одного узгодженого стану в інший узгоджений стан, але всередині транзакції стан даних тимчасово може бути неузгоджений, наприклад, якщо є умова $D = A + B$ – стан узгодженості даних, а під час роботи транзакції значення A та B змінюються, тоді тимчасово умова може виконуватися – $D \neq A + B$ – стан неузгодженості даних, але під час завершення значення цей стан повинен знову стати узгодженим, інакше – помилка;
- *Ізоляція (Isolation)* – результат *CRUD*-операцій однієї транзакції не видно іншим транзакціям до тих пір, поки ця транзакція не завершиться, в свою чергу, ця транзакція не повинна бачити результати *CRUD*-операцій інших транзакцій немов цих транзакцій не існує;
- *Довговічність (Durability)* - як тільки транзакція успішно завершилася, всі результати її *CRUD*-операцій залишаються незмінними і не можуть загубитися.

Нехай O_{ij} – j -та операція i -ї транзакції.

Тоді одну транзакцію можна представити як:

$$Tl = O_{l1}, O_{l2}, O_{l3}, \dots, O_{ln}$$

Атомарність забезпечує два стани, в які може перейти транзакція:

- стан успішного завершення через виконання операції «зафіксувати» (*COMMIT*);
- стан відміни всіх операцій через виконання операції «відмінити» (*ROLLBACK* або *ABORT*).

Якщо операція O_{ij} завершилася з помилкою, тоді транзакція повинна відмінити результати попередніх операцій.

З урахуванням можливих станів внесемо додаткові позначки:

- C_i – позначає операцію фіксацію всіх змін з даними без можливості відміни;
- A_i – позначає операцію відміни всіх змін з даними, виконаних попередніми операціями.

Операції *Read* та *Write* можуть бути записані у скороченій нотації:

- " $wi[B]$ " позначає *Write*-операцію i -ї транзакції у змінну B (у такий спосіб дані модифікуються);
- " $ri[B]$ " позначає *Read*-операцію i -ї транзакції зі змінної B .

Узгодженість стану даних більш чітко можна пояснити на прикладі транзакції з переносу суми грошей з банківського рахунку B на банківський рахунок D .

В таблиці 1 наведено приклад такої транзакції $T1$.

Таблиця 1 – Приклад транзакції

№	Операції $T1$	Значення
1	$r1[B]$	$B := 10$
2	$B := B - 5$	$B := 5$
3	$w1[B]$	$B := 5$
4	$r1[D]$	$D := 10$
5	$D := D + 5$	$D := 15$
6	$w1[D]$	$D := 15$
7	CI	

Нехай спочатку на рахунках B та D була однакова сума грошей = 10. На початку роботи транзакції стан даних на цих рахунках був узгоджений. Але після кроку 3 стан даних став неузгодженим, тому що на рахунку B вже сума зменшилася, а на рахунку D сума ще не збільшилася. Тільки після кроку 6 стан даних знову буде узгоджений.

1.2 Приклади виникаючих проблем при одночасному виконанні транзакцій

При одночасній, паралельній роботі транзакцій кожна операція, як процес ОС, виконується на процесорі на основі витісняючої мультизадачності, по черзі змінюючи одна одну. Але на рівні самих транзакцій та на рівні користувача, який їх розпочав, існує уявлення, що транзакції виконуються одночасно на процесорі.

Для транзакцій робота є напівпаралельною або *квазіпаралельною*.

Для квазіпаралельної роботи транзакцій введено поняття *історії виконання транзакцій H* .

Нехай одночасно розпочали свою роботу дві транзакції $T1, T2$ зі своїми операціями:

$T1 = O11, O12, O13$

$T2 = O21, O22, O23$

Тоді історія $H_{T1,T2} = O11, O21, O12, O22, O13, O23$

Квазіпаралельна історія може бути:

- успішною, коли всі транзакції успішно завершують свою роботу, переходячі до стану фіксації змін, виконаних операціями;
- частково-успішною, коли одна та більше транзакцій переходять до стану відміни своїх операцій.

Квазіпаралельна робота транзакцій може призвести до суперечливого стану даних у вигляді так званих феноменів (*PHENOMENA*):

- феномен "Брудне Читання" ("*Dirty Read*"), в подальшому *P1*;
- феномен "Брудна Модифікація" ("*Dirty Write*"), в подальшому *P2*;
- феномен "Неповторне Читання" ("*Non-repeatable Read*"), в подальшому *P3*.

Враховуючи введені позначки, феномени *P1*, *P2*, *P3* можуть бути представлені як заборона на послідовність операцій, яка призводить до такої історії виконання транзакцій:

- *P1*: $w1[x] \dots r2[x] \dots$ (*a1* і *c2* у будь-якому порядку)
- *P2*: $w1[x] \dots w2[x] \dots$ (*a1 AND c2* у будь-якому порядку)
- *P3*: $r1[P] \dots w2[y \text{ in } P] \dots a2 \dots r1[P] \dots c1$

Наведемо приклади виникання феноменів квазіпаралельного виконання транзакцій.

В таблиці 2 наведено приклад феномену *P1* ("*Dirty Read*") для двох транзакцій *T1*, *T2*.

Таблиця 2 – Приклад феномену *P1* ("*Dirty Read*")

N	Операції <i>T1</i>	Операції <i>T2</i>	Значення
1	$r1[B]$		$B := 10$
2	$B := B - 5$		$B := 5$
3	$w1[B]$		$B := 5$
4		$r2[B]$	$B := 5$
5	$A1$		$B := 10$

Як видно з таблиці 2, в результаті роботи всіх операцій, на 5-му кроці транзакція *T2* вже виконала "*Dirty Read*" - операцію $r2[B]$, тому що значення, яке вона читала на 4-му кроці, вже не існує.

В таблиці 3 наведено приклад феномену *P2* ("*Dirty Write*") для двох транзакцій *T1*, *T2*.

Таблиця 3 – Приклад феномену *P2* ("*Dirty Write*")

N	Операції <i>T1</i>	Операції <i>T2</i>	Значення
1	$r1[B]$		$B := 10$
2	$B := B - 5$		$B := 5$
3	$w1[B]$		$B := 5$
4		$r2[B]$	$B := 5$
5		$B := B - 5$	$B := 0$
6		$w2[B]$	$B := 0$
7	$A1$		$B := 10$

Як видно з таблиці 3, в результаті роботи всіх операцій, на 7-му кроці транзакція $T2$ вже виконала "Dirty Write"-операцію $w2[B]$, тому що значення, яке вона змінила читала на 6-му кроці, вже не існує.

В таблиці 4 наведено приклад феномену $P3$ ("Non-repeatable Read") для двох транзакцій $T1, T2$:

$T1$ – транзакція переносу суми з банківського рахунку B на рахунок D ;

$T2$ – транзакція підрахунку балансу на рахунках B та D (суми грошей на рахунках).

Таблиця 4 – Приклад феномену $P3$ ("Non-repeatable Read") для двох транзакцій $T1, T2$

N	Операції $T1$	Операції $T2$	Значення
1	$r1[B]$		$B := 10$
2	$B := B - 5$		$B := 5$
3	$w1[B]$		$B := 5$
4		$r2[B]$	$B := 5$
5		$r2[D]$	$D := 10$
6	$r1[D]$		$D := 10$
7	$D := D + 5$		$D := 15$
8	$w1[D]$		$D := 15$
9	$A1$		$B := 10, D := 10$
10		$C2$	

Як видно з таблиці 4, на 4-му та 5-му кроках дані знаходяться в неузгодженому стані і транзакція $T2$ отримає неправильний результат підрахунку балансу B та D .

1.3 Протидія виникненню проблем квазіпаралельного виконання транзакцій

1.3.1 Протокол блокування та його операції

Коли йде мова про *Транзакцію* як документ з текстом ділової угоди між декількома учасниками, тоді може розглядатися *Протокол* - документ, в якому фіксуються всі дії в хронологічному порядку, які призвели до успішного створення транзакції.

В інформаційних технологіях протокол – це алгоритм взаємодії двох та більше процесів.

При обміні мережевими повідомленнями між процесами використовується *Комунікаційний протокол* - обумовлені наперед правила передачі даних.

Для боротьби з наслідками квазіпаралельного виконання транзакцій у вигляді феноменів пропонується *Протокол блокування*.

Введемо дві додаткові операції:

$Si[B]$ – операція загального блокування (*Shared lock*) змінної B , яка виконується i -ю транзакцією;

$Xi[B]$ – операція монопольного блокування (*eXclusive lock*) змінної B , яка виконується i -ю транзакцією.

Додатково введемо операцію Ui – зняття всіх блокувань, що встановлені транзакцію Ti .

1.3.2 Протокол 1-го ступеня блокування

Протокол 1-го ступеня блокування містить правила:

- 1) перед операцією $wi[B]$ включається запит на блокування $Xi[B]$
- 2) запит на блокування приймається, якщо він сумісний з вже встановленими блокуваннями B з боку інших транзакцій;
- 3) запити на блокування однієї транзакції завжди сумісні;
- 4) сумісність блокувань визначається матрицею сумісності, приклад якої представлено в таблиці 5;
- 5) якщо запит на блокування не сумісний з вже встановленими блокуваннями інших транзакцій, цей запит переходить у стан очікування (*Wait*), а транзакція перестає виконуватися і також чекає;
- 6) після виконанням операції фіксації або відміни операцій транзакція знімає всі свої встановлені блокування, виконуючи операцію U ;
- 7) після завершення будь-якої транзакції перевіряються всі запити на блокування, які ще чекають.

Таблиця 5 - Матриця сумісності блокувань

$T1/T2$	X	S	-
X	-	-	+
S	-	+	+
-	+	+	+

Символ + в таблицю вказує на сумісність блокувань двох транзакцій.

Для забезпечення контролю запитів на блокування використовується таблиця блокувань, яка містить три колонки:

- назва змінної з бази даних (таблиці);
- перелік встановлених блокувань;
- перелік запитів на блокування.

Протокол 1-го ступеня виключає виникнення феномену "Брудний Запис".

В таблиці 6 наведено приклад роботи протоколу 1-го ступеня блокування.

Таблиця 6 – Приклад роботи протоколу 1-го ступеня блокування

N	Операції $T1$	Операції $T2$	Значення
1	$r1[B]$		$B := 10$
2	$B := B - 5$		$B := 5$
3	$X1[B]$		
4	$w1[B]$		$B := 5$
5		$r2[B]$	$B := 5$
6		$B := B - 5$	$B := 0$
7		$X2[B]$	<i>Wait</i>
8	$A1$		$B := 10$
9	$U1$		
10		$X2[B]$	
11		$W2[B]$	$B := 0$
12		$C2$	
13		$U2$	

На 3-му кроці транзакція $T1$ у відповідності з протоколом виконує запит на блокування $X1[B]$. На поточний момент змінна B не має інших блокувань, тому запит на блокування успішно виконується.

На 7-му кроці транзакція $T2$ у відповідності з протоколом виконує запит на блокування $X2[B]$. Але цей запит не сумісний із вже встановленим блокуванням транзакції $T1$, тому він переходить у стан очікування. На 7-му кроці таблиця блокувань буде містити значення ка показано в таблиці 7.

Таблиця 7 – Приклад заповнення таблиці блокувань

Назва змінної	Перелік встановлених блокувань	Перелік запитів на блокування
B	$X1$	$X2$

Після зняття блокувань транзакції $T1$ повторно виконується запит на блокування на 10-му кроці, який успішно встановлюється. Таким чином протокол 1-го ступеня за рахунок блокування виключає виникнення феномену "Брудний Запис". Але цей протокол не виключає феномен "Брудне Читання" як це видно на 5-му кроці таблиці 6.

Наведемо приклад створення історії квазіпаралельного успішного виконання двох транзакцій з урахування протоколу 1-го ступеню блокувань.

Припустимо є дві транзакції:

$$T1 = r1[B] \ w1[B] \ C1$$

$$T2 = r2[B] \ w2[B] \ C2$$

Історія квазіпаралельного успішного виконання транзакцій для протоколу 1-го ступеня блокування:

$$H(T1, T2) = r1[B] \ r2[B] \ X1[B] \ w1[B] \ X2[B] - Wait \ C1 \ U1 \ X2[B] \ w2[B] \ C2 \ U2$$

1.3.3 Протокол 2-го ступеня блокування

Виключення феномену "Брудне Читання" можливе через додавання S -блокування перед операцією читання, що реалізовано в протоколі 2-го ступеня.

Протокол 2-го ступеня блокування:

- використовує всі правила з протоколу 1-го ступеня
- додаткове правило протоколу: перед операцією $ri[B]$ включається запит на блокування $Si[B]$.

В таблиці 8 наведено приклад роботи протоколу 2-го ступеня блокування.

Таблиця 8 – Приклад роботи протоколу 2-го ступеня блокування

N	Операції $T1$	Операції $T2$	Значення
1	$S1[B]$		$B := 10$
2	$r1[B]$		$B := 10$
3	$B := B - 5$		$B := 5$
4	$X1[B]$		$B := 5$
5	$w1[B]$		$B := 5$
6		$S2[B]$	$Wait$
7	$A1$		$B := 10$
8	$U1$		
9		$S2[B]$	
10		$r2[B]$	$B := 10$
11		$B := B - 5$	$B := 5$
12		$X2[B]$	
13		$W2[B]$	$B := 5$
14		$C2$	
15		$U2$	

Як вказувалось в описі протоколу, запити на блокування однієї транзакції завжди сумісні, тому запит на 4-му кроці є сумісним з вже встановленим блокуванням S .

Наведемо приклад створення історії квазіпаралельного частково успішного виконання двох транзакцій з урахування протоколу 2-го ступеню блокувань, під час якого одна з транзакцій відміняє свої дії.

Припустимо є дві транзакції:

$$T1 = r1[B] \ w1[B] \ C1$$

$$T2 = r2[B] \ w2[B] \ C2$$

Історія квазіпаралельного успішного виконання транзакцій для протоколу 2-го ступеня блокування:

$$H(T1, T2) = S1[B] \ r1[B] \ S2[B] \ r2[B] \ X1[B] - Wait \ A2 \ U2 \ X1[B] \ w2[B] \ C1 \ U1$$

Як видно з історії, після відміни всіх операцій транзакцією $T2$ знімаються всі її блокування операцією $U2$, а транзакція $T1$ продовжує виконання своїх операцій.

1.4 Боротьба з *Deadlock*-станами транзакцій

На жаль, використання блокувань призводить до виникнення іншої проблеми – переходу транзакцій у *Deadlock*-стан (глухий кут), коли:

- дві чи більше транзакції одночасно знаходяться в стані очікування;
- для продовження роботи кожна з транзакцій очікує припинення виконання іншої транзакції.

Найчастіше транзакції переходять до *Deadlock*-стану в одному з двох випадків:

- транзакції виконують однакову послідовність операцій *Read* та *Write* з однаковим елементом даних;
- транзакції виконують послідовність операцій *Read* та *Write* з різними елементами даних у протилежному порядку.

Наведемо приклад виникнення *Deadlock*-стану для двох транзакцій, які виконують однакову послідовність операцій *Read* та *Write* з однаковим елементом даних.

Припустимо є дві транзакції, які виконують однакові операції:

$$T1 = r1[B] \ w1[B] \ C1$$

$$T2 = r2[B] \ w2[B] \ C2$$

Історія квазіпаралельного частково успішного виконання транзакцій для протоколу 2-го ступеня блокування:

$$HT1, T2 = S1[B] \ r1[B] \ S2[B] \ r2[B] \ X1[B] - Wait \ X2[B] - Wait - Deadlock$$

Як видно наприкінці історії обидві транзакції чекають зняття блокування, що визначає появу *Deadlock*-стану.

Наведемо приклад виникнення *Deadlock*-стану для двох транзакцій, які виконують послідовність операцій *Read* та *Write* з різними елементами даних у протилежному порядку.

Припустимо є дві транзакції, які виконують однакові операції:

$T1 = r1[B] \ w1[D] \ C1$

$T2 = r2[D] \ w2[B] \ C2$

Історія квазіпаралельного частково успішного виконання транзакцій для протоколу 2-го ступеня блокування:

$H(T1, T2) = S1[B] \ r1[B] \ S2[D] \ r2[D] \ X1[D] - Wait \ X2[B] - Wait$

Як видно наприкінці історії обидві транзакції чекають зняття блокування, що визначає появу *Deadlock*-стану.

В таблиці 9 наведено приклад заповнення таблиці блокувань після виконання вказаної історії транзакцій.

Таблиця 9 – Приклад заповнення таблиці блокувань

Назва змінної	Перелік встановлених блокувань	Перелік запитів на блокування
<i>B</i>	<i>S1</i>	<i>X2</i>
<i>D</i>	<i>S2</i>	<i>X1</i>

Нижче наведено приклад збереження таблиці у вигляді двомірного масиву елементів `{"name", "set_locks", "wait_locks"}`.

Наприклад, на мові програмування *C* такий масив для прикладу з таблиці 9 може мати наступний вигляд:

```
char* lock_table[][3] = {  
  { "B", "S1", "X2" },  
  { "D", "S2", "X1" } };
```

Бажано, щоб при виникненні *Deadlock*-стану СКБД була здатна:

- знайти *Deadlock*-стан;
- вивести транзакції з *Deadlock*-стану.

Для виявлення *Deadlock*-стану варто знайти цикл у діаграмі станів очікування, тобто в переліку "транзакцій, що очікують закінчення виконання інших транзакцій".

Для визначення *Deadlock*-стану взаємного блокування транзакцій створюється граф очікування транзакцій, використовуючи наступні правила:

- 1) кожна транзакція визначає вузол графу (*T1*, *T2*, *T3* ...);

2) між вузлом $T1$ та вузлом $T2$ створюється направлена дуга, якщо транзакція $T1$ намагається встановити блокування, але воно не сумісне з вже встановленими блокуваннями транзакції $T2$.

Якщо у графі буде знайдено цикл, це вказує на наявність *Deadlock*-стану.

На рисунку 1 показано приклад графу для раніше описаної історії транзакцій.

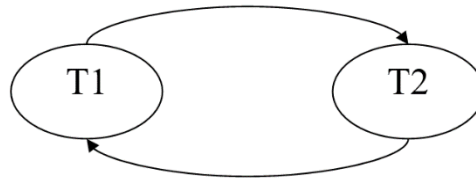


Рис. 1 – Приклад зі графу очікування транзакцій для історії

Граф очікування можна представити у вигляді двовірного масиву елементів $\{i, j\}$, де i, j - номери транзакцій, якщо між вузлом $T1$ та вузлом $T2$ існує направлена дуга.

Наприклад, на мові програмування *C* такий масив для прикладу з рисунку 1 може мати наступний вигляд:

```
int wait[2][2] = {{1, 2}, {2, 1}};
```

Пошук виходу з *Deadlock*-стану полягає у виборі однієї з заблокованих транзакцій як жертви і скасуванні її виконання. Таким чином, з неї знімається блокування, а виконання інших транзакцій може бути відновлено.

1.5 Основи керування базами даних з використанням мови *SQL*

1.5.1 Перевага використання систем керування базами даних

В попередніх розділах розглядалися особливості протоколів блокування елементів даних. Але як ці данні виглядають та яку структуру вони мають?

На мові програмування *C* існує можливість створювати композитні (складні) типи даних через оператор *Struct*. На рисунку 2 наведено приклад зі створення структури «банківський рахунок», об'яви та ініціалізації масиву зі 100 банківських рахунків.

```
struct bill {
    int  n_bill; /* номер рахунку */
    float  bill_count; /* кількість коштів */
    char name[20]; /* прізвище власника рахунку */
};
// об'ява та ініціалізація структури
struct bill bills[100] = {{1, 100, "Ivanenko" } ,
{2, 50, "Petrenko" } , ... };
```

Рис. 2 – Приклад зі створення структури «банківський рахунок»

Створений масив в подальшому можна обробляти в оперативній пам'яті.

Але після завершення роботи програми значення елементів масиву буде втрачено, якщо ці елементи не зберегти в окремому файлі.

На мові програмування C існують функції керування файлами та керування операціями над складними структурами типу *struct*.

Але цей процес потребує додаткових зусиль.

Наприклад, щоб записати у файл таку структуру даних, необхідно:

- створити змінні з посиланнями на дескриптор файлу;
- відкрити файл у режимі запису, якщо файл вже існує, або створити його;
- в циклі записати кожний елемент масиву з структурами;
- закрити файл;
- очистити всі змінні, пов'язані з файлом.

Наприклад, щоб читати із файлу раніше записані дані, необхідно:

- створити змінні з посиланнями на дескриптор файлу;
- відкрити файл у режимі читання;
- в циклі зчитати кожний рядок запис файлу в кожний елемент масиву з структурами;
- закрити файл;
- очистити всі змінні, пов'язані з файлом.

Відомо, що однією з цілей роботи ОС – є зменшення трудомісткості процесу взаємодії з комп'ютером з боку людини.

Для досягнення цієї цілі в процесах керування складними структурами даних було створено спеціальне системне програмне забезпечення для керування операціями обробки таких структур.

Таке програмне забезпечення відноситься до підкласу *систем керування базами даних (СКБД)*.

СКБД повинна спростити людині процеси з керування файлами, в яких є складні структури даних.

База даних, БД (англ. *DataBase*) – будь-який впорядкований набір даних різної структури.

Наприклад, БД можна вважати масив банківських рахунків. Але на відміну від масиву, база даних повинна автоматично зберігатися у файлі.

Сьогодні використовуються різні типи БД, але серед ієрархічних, мережових та об'єктно-орієнтованих баз даних вже майже 50 років окремо виділяється реляційна база даних, де прикметник «реляційна» визначає *Relations* – відношення.

В реляційній БД файли з даними називаються *таблицями* бази даних (*реляційними таблицями*), які можуть бути пов'язані один з одним через стовпчики. Прикладом таблиць бази даних можуть бути файли формату *CSV*, які розглядалися в попередніх лабораторних роботах, наприклад, для файлів */etc/passwd*, */etc/group*, в яких розідбником між стовпчиками є символ : (двокрапка). Серед команд обробки подібних файлів є команда *join*, яка об'єднує два *CSV*-файли за відповідними двома стовпчиками, наприклад, наступна команда об'єднує два *CSV*-файли за 1-м стовпчиком 1-го файлу та 1-м стовпчиком 2-го файлу:

```
sort -n -t: -k 4 /etc/passwd > passwd_sort
sort -n -t: -k 3 /etc/group > group_sort
join -t: -j1 4 -j2 3 -o 1.1 2.1 1.5 ./passwd_sort ./group_sort
```

Ця команда є прикладом об'єднання двох реляційних таблиць через співпадаючі значення двох стовпчиків (вже відоме поняття *Relations*), які представлено у форматі *CSV*-файлу. Але використання цієї команди вже є трудомісткою задачею для програміста.

Мабуть так працювали програмісти, створюючи власні програми керування реляційними БД, які містили багато таблиць, коли компанія *IBM* прийняла рішення про створення експериментальної СКБД *System R*, сконструйованої для демонстрації використання переваг моделі реляційних даних з урахуванням багато користувальницького режиму роботи.

СКБД автоматизує роботу більшості процесів взаємодії з БД:

- створення даних;
- зміна даних;
- видалення даних;
- перегляд даних;
- обмеження доступу до даних;
- керування протоколами блокування при доступі до даних транзакцій.

1.5.2 Мова *SQL* – стандарт керування реляційними базами даних

Мова SQL (англ. *Structured query language*) – мова структурованих запитів, декларативна мова програмування, яка спрощує взаємодію користувача з СКБД. Була вперше впроваджена в СКБД *IBM System R*.

На відміну від імперативних мов програмування, наприклад мови *C*, мова *SQL* може формувати інтерактивні запити, виступати як інструкції для керування даними. Інтерактивний режим забезпечується самою СКБД.

Основними перевагами мови *SQL* є:

– декларативність – протилежність імперативності, яка використовується в алгоритмічних мовах програмування, і дозволяє людині лише вказувати СКБД, що вона хоче зробити і не вказувати як це треба зробити;

– стандартизація – можливість виконувати однакові команди в різних СКБД (*MySQL*, *PostgreSQL*, *MS SQL*, *Oracle*, *IBM DB2*).

На відміну від мови *C* оператори мови *SQL* не чутливі до регістру символів.

Для створення таблиць будь-яких структур даних мова *SQL* використовує оператор *CREATE TABLE*:

```
CREATE TABLE назва_таблиці (  
    змінна_1 тип_змінної_1,  
    змінна_2 тип_змінної_2,  
    ...  
);
```

Важливо пам'ятати, що, за замовчуванням, всі команди мови *SQL* повинні завершатися символом ; (крапка з комою).

Основними типами даних є:

integer – ціле значення;

float – значення плаваючою комою;

char(m) – рядок символів довжиною *m*;

date – дата.

Нижче наведено приклад створення таблиці банківських рахунків:

```
CREATE TABLE bills (  
    n_bill integer,  
    bill_count float,  
    name char(20)  
);
```

Для видалення таблиці мова *SQL* використовує оператор *DROP TABLE*:

```
DROP TABLE назва_таблиці;
```

Нижче наведено приклад видалення таблиці банківських рахунків:

```
DROP TABLE bills;
```

На відміну від масиву банківських рахунків з рисунку 1, таблиця дозволяє зберігати змінну кількість елементів без необхідності попереднього виділення пам'яті.

Для додавання нового рядка в таблицю використовується оператор *INSERT INTO*:

```
INSERT INTO назва_таблиці VALUES (  
    значення_змінної_1,  
    ...  
    значення_змінної_n  
);
```

Строкові типи даних та дати використовують одинарні прямі лапки.

Нижче наведено приклад додавання двох елементів до таблиці банківських рахунків:

```
INSERT INTO bills VALUES (1,100, 'Ivanenko');  
INSERT INTO bills VALUES (2,50, 'Petrenko');
```

Для зміни значень змінних (колонки) рядка таблиці використовується оператор *UPDATE*:

```
UPDATE назва_таблиці SET  
    змінна_1 = значення_змінної_1,  
    ...  
    змінна_n = значення_змінної_n  
WHERE умови_зміни;
```

Опис фрази *умови_зміни* використовує формат:

змінна_i = *значення_змінної_i*

Нижче наведено приклад зміни *bill_count* = 150 для рахунку з *n_bill* = 1:

```
UPDATE bills SET bill_count = 150 WHERE n_bill = 1;
```

В умовах зміни можна включати декілька умов, використовуючи логічні операції *AND* або *OR*.

Нижче наведено приклад зміни *bill_count* = 150 для рахунку з *n_bill* = 1 або 2:

```
UPDATE bills SET bill_count = 150  
    WHERE n_bill = 1 OR n_bill = 2;
```

Для видалення елементів таблиці мова *SQL* використовує оператор *DELETE*:

```
DELETE FROM назва_таблиці  
WHERE умови_зміни;
```

Нижче наведено приклад видалення рахунку з *n_bill* = 1:

```
DELETE FROM bills WHERE n_bill = 1;
```

Для отримання значень змінних таблиці мова *SQL* використовує оператор *SELECT*:

```
SELECT змінна_1, змінна_2, ..., змінна_n  
FROM назва_таблиці  
WHERE умови_відбору;
```

Опис фрази *умови_відбору* визначає елементи таблиці, змінні яких необхідно прочитати, і еквівалентний опису фрази *умови_зміни*.

Нижче наведено приклад отримання значення розмір коштів на рахунку рахунку з $n_bill = 1$:

```
SELECT bill_count FROM bills WHERE n_bill = 1;
```

Якщо необхідно отримати значення всіх змінних, використовується символ `*`.

Наприклад:

```
SELECT * FROM bills WHERE n_bill = 1;
```

1.6 Основи роботи з СКБД *PostgreSQL*

СКБД *PostgreSQL* – розповсюджена СКБД з відкритим *source*-кодом. Прототип був розроблений в Каліфорнійському університеті Берклі в 1987 році під назвою *POSTGRES*, після чого активно розвивався і доповнювався. Подробиці про СКБД *PostgreSQL* на сайті за адресою <https://www.postgresql.org>

Інсталяційні пакети СКБД для різних ОС доступні за адресою <https://www.postgresql.org/download/>.

Після інсталяції СКБД для створення користувачів СКБД *PostgreSQL* в командному рядку використовується утиліта *createuser*:

```
createuser ім'я_користувача --username=USERNAME
```

Але для створення нових користувачів після інсталяції необхідно, щоб користувач на ім'я *USERNAME* мав відповідні права адміністратора, якими за замовчуванням володіє лише користувач *postgres*. Якщо ви виконаєте команди від імені користувача *postgres* як користувача ОС *Linux*, тоді опцію `--username` можна не вказувати. Наприклад, для створення користувача на ім'я *petrenko* можна виконати команду:

```
createuser blazhko
```

Для створення БД в СКБД *PostgreSQL* в командному рядку використовується утиліта *createdb*:

```
createdb ім'я_користувача --username=USERNAME
```

Наприклад, для створення бази даних на ім'я *test_db* можна виконати команду:

```
createdb test_db
```

Для встановлення зв'язку з СКБД *PostgreSQL* в ОС *Linux* використовується утиліта *psql* з основними параметрами:

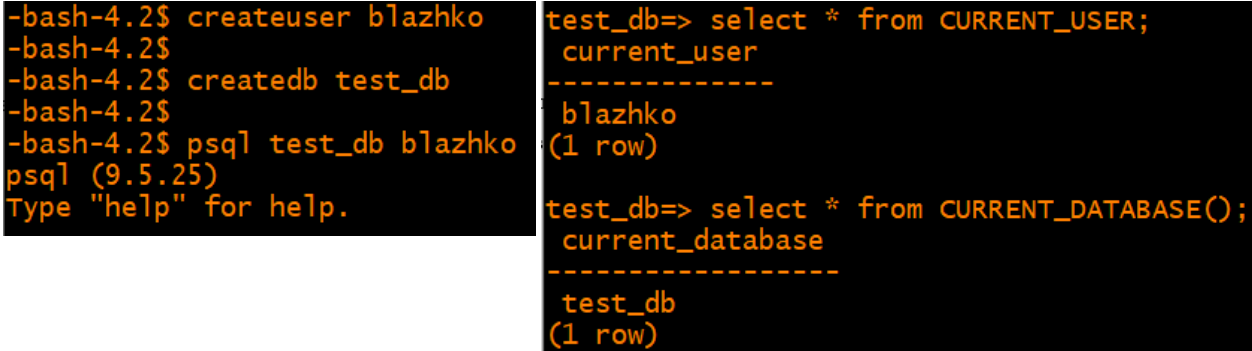
```
psql [-U USERNAME] [DBNAME] або psql [DBNAME] [USERNAME]
```

За замовчуванням значення параметрів *DBNAME* та *USERNAME* співпадає з іменем користувача ОС *Linux*. На рисунку 3 наведено екрану форму терміналу з прикладом

виконання команд створення користувача *blazhko*, створення БД *test_db* та встановлення зв'язку з СКБД *PostgreSQL* для роботи з БД *test_db* від імені користувача *blazhko*.

Для отримання поточних значень імені користувача та назви бази даних можна використати наступні команди:

```
SELECT CURRENT_USER;
SELECT CURRENT_DATABASE();
```



The screenshot shows a terminal session where the user *blazhko* is created, a database *test_db* is created, and the user connects to *test_db* using *psql*. The user then runs two SQL queries: `select * from CURRENT_USER;` which returns *blazhko*, and `select * from CURRENT_DATABASE();` which returns *test_db*.

Рис. 3 – Приклад виконання команд

Для завершення роботи з утилітою використовується команда *\q* або *quit*.

1.7 Робота з транзакціями в СКБД

В СКБД *PostgreSQL* для початку роботи транзакції використовується оператор *START TRANSACTION*;

Наведемо приклад транзакції, яка містить операції, описані в таблиці 10.

Таблиця 10 - Приклад команд транзакції з операцією фіксації

№	Команда	Опис
1	<i>START TRANSACTION;</i>	Розпочати транзакцію
2	<i>SELECT bill_count FROM bills WHERE n_bill = 1;</i>	Прочитати розмір коштів на рахунку з <i>n_bill = 1</i>
3	<i>UPDATE bills SET bill_count = 150 WHERE n_bill = 1;</i>	Змінити розмір коштів на рахунку з <i>n_bill = 1</i>
4	<i>SELECT bill_count FROM bills WHERE n_bill = 1;</i>	Повторно прочитати розмір коштів на рахунку з <i>n_bill = 1</i>
5	<i>COMMIT;</i>	Зафіксувати зміни у файлі БД
6	<i>ROLLBACK;</i>	Спроба відмінити операції
7	<i>SELECT bill_count FROM bills WHERE n_bill = 1;</i>	Повторно прочитати розмір коштів на рахунку з <i>n_bill = 1</i>

На рисунку 4 наведено приклади виконання транзакції.

```
-bash-4.2$ psql
psql (9.5.25)
Type "help" for help.

postgres=# START TRANSACTION;
START TRANSACTION
postgres=# SELECT bill_count FROM bills WHERE n_bill = 1;
 bill_count
-----
         200
(1 row)

postgres=# UPDATE bills SET bill_count = 150 WHERE n_bill = 1;
UPDATE 1
postgres=# commit;
COMMIT
postgres=# |
```

Рис. 4 – приклади виконання транзакції з операціями з таблиці 1

У відповідності із атомарністю транзакцій після операції *COMMIT* вже не можливо відмінити операції. Це підтвердить результат виконання операції на 7-му кроці таблиці 10.

У відповідності із атомарністю транзакцій після операції *ROLLBACK* всі операції відміняються, що підтвердить результат виконання операції на 6-му кроці таблиці 11.

Таблиця 11 - Приклад команд транзакції з операцією відміни

№	Команда	Опис
1	<i>START TRANSACTION;</i>	Розпочати транзакцію
2	<i>SELECT bill_count FROM bills WHERE n_bill = 1;</i>	Прочитати розмір коштів на рахунку з <i>n_bill = 1</i>
3	<i>UPDATE bills SET bill_count = 150 WHERE n_bill = 1;</i>	Змінити розмір коштів на рахунку з <i>n_bill = 1</i>
4	<i>SELECT bill_count FROM bills WHERE n_bill = 1;</i>	Повторно прочитати розмір коштів на рахунку з <i>n_bill = 1</i>
5	<i>ROLLBACK;</i>	Відмінити зміни у файлі БД
6	<i>SELECT bill_count FROM bills WHERE n_bill = 1;</i>	Повторно прочитати розмір коштів на рахунку з <i>n_bill = 1</i>

При необхідності можна виконати ще одну транзакцію квазіпаралельно, коли операції двох транзакцій будуть виконуватися одна за одною, як показано в таблиці 12.

Таблиця 12 - Приклад виконання команд двох транзакцій

№	Команда T1	Команда T2
1	<i>START TRANSACTION;</i>	
2		<i>START TRANSACTION;</i>
3	<i>SELECT bill_count FROM bills WHERE n_bill = 1;</i>	
4		<i>SELECT bill_count FROM bills WHERE n_bill = 2;</i>
5	<i>UPDATE bills SET bill_count = 150 WHERE n_bill = 1;</i>	
6		<i>UPDATE bills SET bill_count = 120 WHERE n_bill = 2;</i>
7	<i>COMMIT;</i>	
8		<i>COMMIT;</i>

На рисунку 5 показано приклад двох транзакцій з таблиці 10, які виконуються у двох окремих псевдотерміналах ОС *Linux*.

```

root@vpsj3leQ:~
-bash-4.2$ psql
psql (9.5.25)
Type "help" for help.

postgres=# START TRANSACTION;
START TRANSACTION
postgres=# SELECT bill_count FROM bills WHERE n_bill = 1;
 bill_count
-----
         150
(1 row)

postgres=#

```

```

root@vpsj3leQ:~
-bash-4.2$ psql
psql (9.5.25)
Type "help" for help.

postgres=# START TRANSACTION;
START TRANSACTION
postgres=# SELECT bill_count FROM bills WHERE n_bill = 1;
 bill_count
-----
         150
(1 row)

postgres=#

```

Рис. 5 – Приклади виконання транзакції з операціями з таблиці 10

1.8 Робота протоколу блокування в СКБД *PostgreSQL*

В СКБД *PostgreSQL* операції базового протоколу блокування виконуються на основі оператора *LOCK TABLE*, синтаксис якого описано в таблиці 13.

Таблиця 13 - Формати операцій встановлення блокування в СКБД *PostgreSQL*.

№	Формат операції встановлення блокування	Тип блокування
1	<i>LOCK TABLE назва_таблиці IN EXCLUSIVE MODE [NOWAIT];</i>	<i>X</i>
2	<i>LOCK TABLE назва_таблиці IN SHARE MODE [NOWAIT];</i>	<i>S</i>

Як вже вказувалось в теорії блокування, якщо запит на операцію блокування транзакції *T2* не сумісний із вже встановленим блокуванням транзакції *T1*, операція чекає на зняття цього блокування.

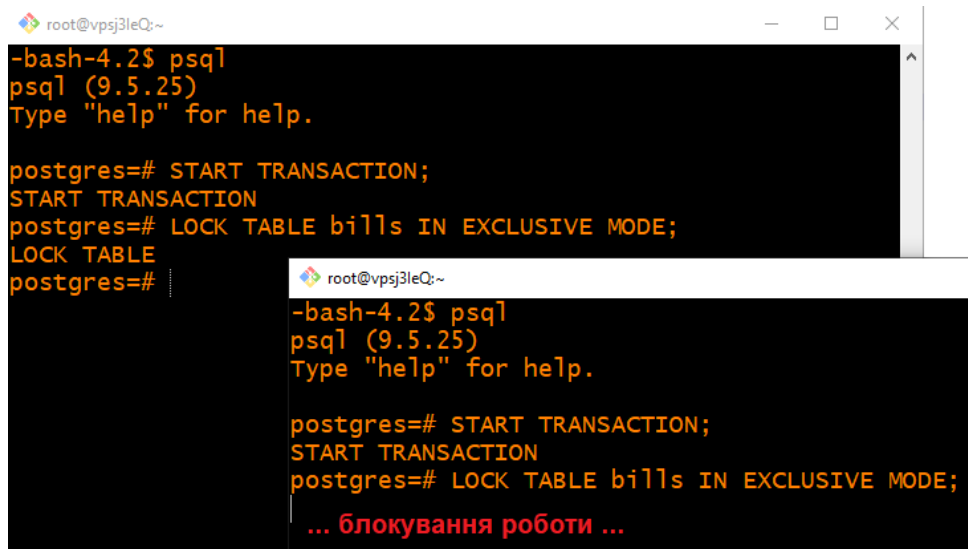
Якщо команда *LOCK TABLE* не повинна чекати на зняття блокування, тоді до команди додається опція *NOWAIT*.

В таблиці 14 наведено приклади виконання транзакцій з використанням операцій блокування.

Таблиця 14 - Приклад виконання команд двох транзакцій з протоколом блокування

№	Команди T1	Команди T2
1	<i>START TRANSACTION;</i>	
2		<i>START TRANSACTION;</i>
3	<i>LOCK TABLE bills</i> <i>IN EXCLUSIVE MODE;</i>	
5		<i>LOCK TABLE bills IN EXCLUSIVE MODE;</i>
6		блокування операції та перехід у стан очікування внаслідок несумісності блокування
7	<i>COMMIT;</i>	
11		Завершення очікування
12		<i>COMMIT;</i>

На рисунку 6 представлено приклад виконання операцій двох транзакцій, коли у відповідності з кроком 5 транзакція *T2* (права частина екрану) виконує запит на блокування, не сумісний з запитом на блокування транзакції *T1* (ліва частина екрану). Операція блокування транзакції *T2* переходить у стан очікування.



```
root@vpsj3leQ:~  
-bash-4.2$ psql  
psql (9.5.25)  
Type "help" for help.  
  
postgres=# START TRANSACTION;  
START TRANSACTION  
postgres=# LOCK TABLE bills IN EXCLUSIVE MODE;  
LOCK TABLE  
postgres=#  
  
root@vpsj3leQ:~  
-bash-4.2$ psql  
psql (9.5.25)  
Type "help" for help.  
  
postgres=# START TRANSACTION;  
START TRANSACTION  
postgres=# LOCK TABLE bills IN EXCLUSIVE MODE;  
... блокування роботи ...
```

Рис. 6 – Приклади виконання транзакції з операціями блокування таблиці *bills*

1.9 Аналіз процесів СКБД та особливості взаємовідносин *psql*-процесу та СКБД-процесу

Утиліта *psql* – це звичайний процес в ОС *Linux*, який взаємодіє з процесом СКБД. Коли *psql* з'єднується з СКБД автоматично створюється копія серверного процесу *postgres*.

На рисунку 7 показано результати команди *ps -u postgres -o pid,ppid,stat,cmd*

Нижче надано коментарі щодо запущених процесів:

- процес з *PID=31552* – це основний СКБД-процес – *daemon*-процес на ім'я *postmaster*, який запускається, наприклад, через виконання команди */usr/bin/postmaster -D шлях_до_каталогу_БД*;

- процес *postmaster* створив 7 *child*-процесів, які виконують системні функції керування базою даних;

- процес з *PID=92115* – це процес *psql*, ініційований відповідною командою *psql*.

- процес *postgres* (*PID=92116*) – копію серверного процесу, яку створено під час першого встановлення з'єднання процесу *psql* із основним серверним процесом.

PID	PPID	STAT	CMD
31552	1	Ss	/usr/bin/postmaster -D /var/lib/pgsql/data
31554	31552	Ss	postgres: logger
31556	31552	Ss	postgres: checkpointer
31557	31552	Ss	postgres: background writer
31558	31552	Ss	postgres: walwriter
31559	31552	Ss	postgres: autovacuum launcher
31560	31552	Ss	postgres: stats collector
31561	31552	Ss	postgres: logical replication launcher
92059	1	Ss	/usr/lib/systemd/systemd --user
92062	92059	S	(sd-pam)
92068	92056	S	sshd: postgres@pts/1
92069	92068	Ss	-bash
92115	92069	S+	psql
92116	31552	Ss	postgres: postgres postgres [local] idle

Рис. 7 – Результати команди *ps -u postgres -o pid,ppid,stat,cmd*

Процес *psql* та *postgres* завжди очікують виконання наступних операцій транзакцій.

Процес блокування дії інших транзакцій може стати небезпечним для роботи системи при виникненні позаштатних ситуацій. Наведемо приклад такої ситуації:

- в першому терміналі виконується комбінація клавіш *<Ctrl>+<Z>* як еквівалент передачі поточному процесу команди *TSTR* - сигналу зупинки процесу;
- в другому терміналі транзакція *T2* виконує команду блокування, яка чекає на завершення транзакції *T1* з несумісним блокуванням.

На рисунку 8 представлено стан процесів після виконання вказаних команд *LOCK TABLE*, коли в процесі транзакції *T2* показано операцію, яка продовжує очікувати завершення транзакції *T1*. І таке очікування може бути будь-якої тривалості.

```
28740 27602 S+  psql
28742 23206 Ss  postgres: postgres postgres [local] idle in transaction
30142 27890 S+  psql
30144 23206 Ss  postgres: postgres postgres [local] LOCK TABLE waiting
[oracle@vpsj3IeQ ~]$
```

Рис. 8 – Фрагмент результату команди *ps*

Для завершення процесу очікування можливі два варіанти:

- нормальне завершення транзакції *T1*;
- ліквідація процесу транзакції *T1* з використанням команди *kill*.

Якщо транзакція не хоче чекати на завершення роботи іншої транзакції, вона може виконувати команду блокування з додатковим параметром *NOWAIT*, тоді при несумісності блокування буде видано повідомлення про помилку, приклад якої наведено на рисунку 9: *could not obtain lock on relation "bills"* – не можу встановити блокування на таблицю *"bills"*

```
-bash-4.2$ psql
psql (9.5.25)
Type "help" for help.

postgres=# START TRANSACTION;
START TRANSACTION
postgres=# LOCK TABLE bills IN EXCLUSIVE MODE;
LOCK TABLE
postgres=#
```

```
root@vpsj3leQ:~
-bash-4.2$ psql
psql (9.5.25)
Type "help" for help.

postgres=# START TRANSACTION;
START TRANSACTION
postgres=# LOCK TABLE bills IN EXCLUSIVE MODE NOWAIT;
ERROR:  could not obtain lock on relation "bills"
postgres=#
```

Рис. 9 – Приклад помилки блокування

1.10 Боротьба з *Deadlock*-станами в СКБД *PostgreSQL*

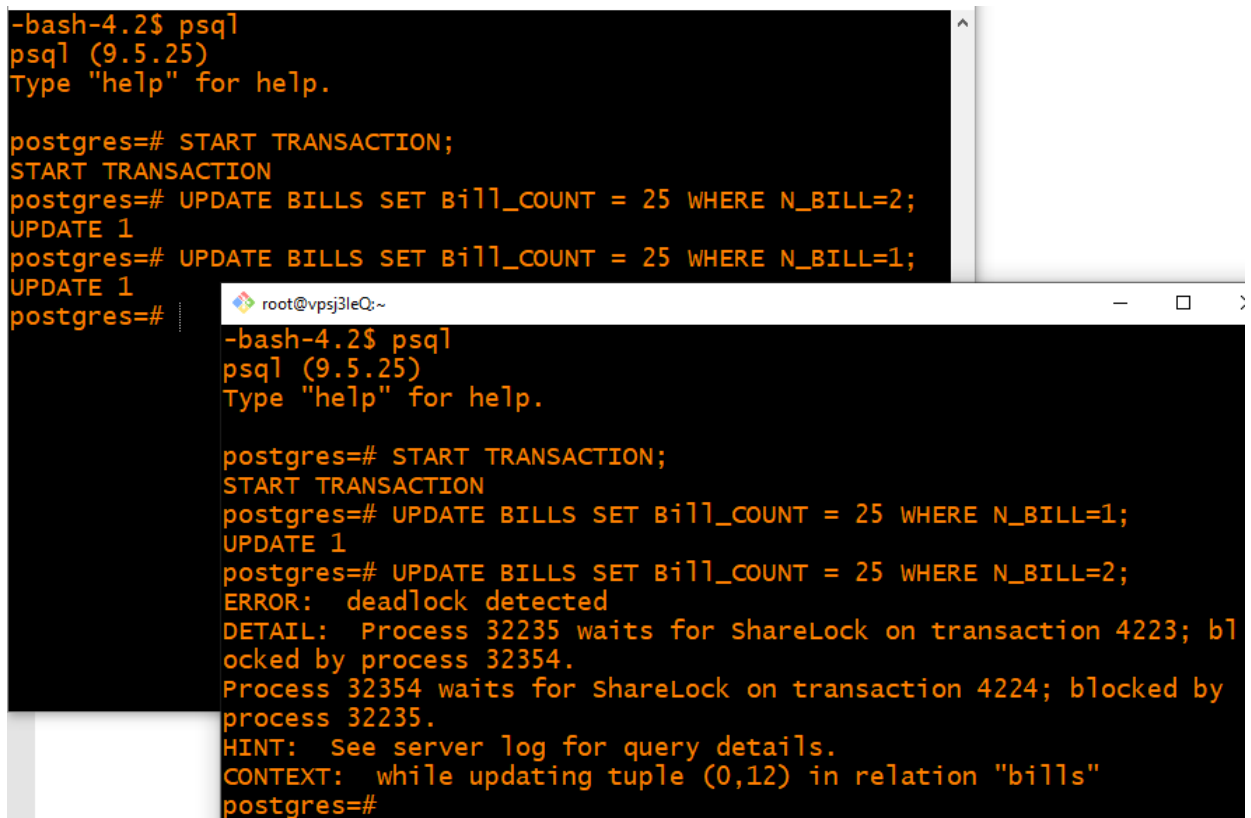
При виникненні *Deadlock*-стану у виконанні транзакції СКБД *PostgreSQL* приймає рішення про відміну транзакції, операція якої призвела до цього стану.

В таблиці 15 наведено приклад виконання команд двох транзакцій, які призводять до *Deadlock*-стану.

Таблиця 15 - Приклад виконання команд двох транзакцій, які призводять до *Deadlock*-стану.

№	SQL-операція транзакції № 1	SQL- операція транзакції № 2
1.	<i>START TRANSACTION;</i>	
2.		<i>START TRANSACTION;</i>
3.	<i>UPDATE BILLS SET Bill_COUNT = 25 WHERE N_BILL=2;</i>	
4.		<i>UPDATE BILLS SET Bill_COUNT = 25 WHERE N_BILL=1;</i>
5.	<i>UPDATE BILLS SET Bill_COUNT = 25 WHERE N_BILL=1;</i>	
6.		<i>UPDATE BILLS SET Bill_COUNT = 25 WHERE N_BILL=2;</i>

Результати роботи команд представлено на рисунку 10.



```
-bash-4.2$ psql
psql (9.5.25)
Type "help" for help.

postgres=# START TRANSACTION;
START TRANSACTION
postgres=# UPDATE BILLS SET Bill_COUNT = 25 WHERE N_BILL=2;
UPDATE 1
postgres=# UPDATE BILLS SET Bill_COUNT = 25 WHERE N_BILL=1;
UPDATE 1
postgres=#
```

```
root@vpsj3IeQ:~
-bash-4.2$ psql
psql (9.5.25)
Type "help" for help.

postgres=# START TRANSACTION;
START TRANSACTION
postgres=# UPDATE BILLS SET Bill_COUNT = 25 WHERE N_BILL=1;
UPDATE 1
postgres=# UPDATE BILLS SET Bill_COUNT = 25 WHERE N_BILL=2;
ERROR:  deadlock detected
DETAIL:  Process 32235 waits for ShareLock on transaction 4223; blocked by process 32354.
Process 32354 waits for ShareLock on transaction 4224; blocked by process 32235.
HINT:  See server log for query details.
CONTEXT:  while updating tuple (0,12) in relation "bills"
postgres=#
```

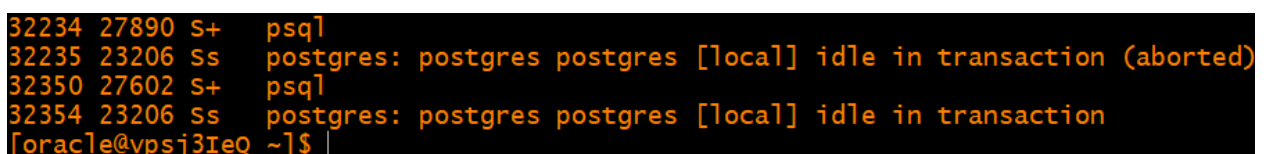
Рис. 10 – Результати роботи команд двох транзакцій, які призводять до *Deadlock*-стану

Різні СКБД по-різному реагують на появу *Deadlock*-станів. СКБД *PostgreSQL* скасувала другу транзакцію, команда якої привела до *Deadlock*-стану, а першу транзакцію зберегла. Тобто СКБД *PostgreSQL* автоматично скасовує транзакцію, остання операція якої призвела до появи *Deadlock*-стану. Скасована транзакція вже не зможе продовжити свою роботу. Єдине, що вона зможе зробити – це виконати команду *ROLLBACK*, щоб дати можливість в терміналі запустити нову транзакцію, інакше її подальші дії будуть ігноруватися СКБД з відповідним повідомленням про помилку.

Як видно з рисунку 10, транзакція *T2* отримала повідомлення про помилку, яка пов'язана з *Deadline*-станом: *ERROR: deadlock detected*
DETAIL: Process 32235 waits for ShareLock on transaction 4223; blocked by process 32354.
Process 32354 waits for ShareLock on transaction 4224; blocked by process 32235.

В повідомленні вказані *PID* процесів, в яких виконуються транзакції.

Для аналізу результатів роботи транзакції додатково переглянемо таблицю процесів, фрагмент якої зображено на рисунку 11.



```
32234 27890 S+   psql
32235 23206 ss   postgres: postgres postgres [local] idle in transaction (aborted)
32350 27602 S+   psql
32354 23206 ss   postgres: postgres postgres [local] idle in transaction
[oracle@vpsj3IeQ ~]$
```

Рис. 11 – Фрагмент таблиці процесів, отриманий командою *ps*

2 Завдання лабораторної роботи

Виконати наступні дії з підготовки до виконання завдань роботи:

- 1) встановити з'єднання з віддаленим *Linux*-сервером з IP-адресою = 46.175.148.116, логіном, наданим вам викладачем, та паролем, зміненим вами у попередній роботі;
- 2) перейти до каталогу *Git*-репозиторія;
- 3) створити нову *Git*-гілку з назвою «*Laboratory-work-10*»;
- 4) перейти до роботи зі створеною гілкою;
- 5) створити каталог з назвою «*Laboratory-work-10*»;
- 6) перейти до каталогу «*Laboratory-work-10*»;
- 7) в каталозі «*Laboratory-work-10*» створити файл *README.md* та додати до файлу рядок тексту із темою лабораторної роботи «*Керування процесами-транзакціями в базах даних*» як заголовок 2-го рівня *Markdown*-форматування;
- 8) виконати операції з оновлення *GitHub*-репозиторію змінами *Git*-репозиторія через послідовність *Git*-команд *add*, *commit* із коментарем «*Changed by Local Git*» та *push*;
- 9) на веб-сервісі *GitHub* перейти до створеної гілки «*Laboratory-work-10*»;
- 10) перейти до каталогу «*Laboratory-work-10*» та розпочати процес редагування файлу *README.md* ;
- 11) в подальшому за результатами рішень кожного наступного розділу завдань до файлу *README.md* додавати рядки як заголовки 3-го рівня *Markdown*-форматування з назвами розділу та знімки екрані з підписами до знімків екранів з описом пунктів завдань (для завдань 2.2., 2.3, 2.4).

2.1 Теоретичне завдання зі створення історій виконання транзакцій

Припустимо, що існують три транзакції, приклади яких представлено в таблиці 15.

У відповідності з вашим варіантом виконати наступні теоретичні завдання, представивши рішення у файлі *README.md* у вигляді рядків звичайного тексту. Вказати у файлі також початкові дані з описом операцій транзакцій.

2.1.1 Створити історію квазіпаралельного успішного виконання транзакцій для протоколу 1-го ступеня блокування.

Описати таблицю блокування транзакцій до моменту першої фіксації змін однієї з транзакцій у вигляді масиву на мові програмування *C*.

2.1.2 Для створеної історії з попереднього завдання визначити наявність *Deadlock*-стану транзакції, створивши граф очікування транзакцій, та зробити відповідний висновок.

Представити граф у вигляді масиву на мові програмування *C*.

2.1.3 Створити історію квазіпаралельного успішного виконання транзакцій для протоколу 2-го ступеня блокування.

Описати таблицю блокування транзакцій до моменту першої фіксації змін однієї з транзакцій у вигляді масиву на мові програмування C.

2.1.4 Для створеної історії з попереднього завдання визначити наявність *Deadlock*-стану транзакції, створивши граф очікування транзакцій, та зробити відповідний висновок.

Представити граф у вигляді масиву на мові програмування C.

Таблиця 15 – Варіанти завдань з прикладами транзакцій

№ вар.	Приклади транзакцій
1	$T1 = R[A] R[B] W[A] C1$ $T2 = W[D] R[B] W[B] C2$ $T3 = R[D] W[D] C3$
2	$T1 = W[A] R[B] W[B] C1$ $T2 = R[D] R[B] W[D] C2$ $T3 = R[A] W[A] C3$
3	$T1 = R[A] W[A] C1$ $T2 = R[D] R[B] W[D] C2$ $T3 = W[A] R[B] W[A] C3$
4	$T1 = W[A] W[D] C1$ $T2 = R[D] R[B] W[D] C2$ $T3 = W[A] R[B] W[D] C3$
5	$T1 = W[A] W[B] C1$ $T2 = R[D] R[A] W[D] C2$ $T3 = W[A] R[B] W[D] C3$
6	$T1 = W[D] W[A] C1$ $T2 = R[D] R[A] W[D] C2$ $T3 = W[A] R[B] W[D] C3$
7	$T1 = W[D] W[A] C1$ $T2 = R[D] R[A] W[D] C2$ $T3 = W[B] R[A] W[D] C3$
8	$T1 = W[D] R[B] W[B] C1$ $T2 = R[A] R[B] W[A] C2$ $T3 = R[D] W[D] C3$
9	$T1 = R[D] R[B] W[D] C1$ $T2 = W[A] R[B] W[B] C2$ $T3 = R[A] W[A] C3$

Таблиця 15 – Продовження

№ вар.	Приклади транзакцій
10	$T1 = R[D] R[B] W[D] C1$ $T2 = R[A] W[A] C2$ $T3 = W[A] R[B] W[A] C3$
11	$T1 = W[D] R[B] W[B] C1$ $T2 = R[A] R[B] W[A] C2$ $T3 = R[D] W[D] C3$
12	$T1 = R[A] W[A] C1$ $T2 = W[A] R[B] W[B] C2$ $T3 = R[D] R[B] W[D] C3$
13	$T1 = R[A] W[A] C1$ $T2 = R[D] R[B] W[D] C2$ $T3 = W[A] R[B] W[A] C3$
14	$T1 = W[A] W[D] C1$ $T2 = R[D] R[B] W[D] C2$ $T3 = W[A] R[B] W[D] C3$
15	$T1 = W[D] W[A] C1$ $T2 = R[D] R[A] W[D] C2$ $T3 = W[A] R[B] W[D] C3$
16	$T1 = W[D] W[A] C1$ $T2 = R[D] R[A] W[D] C2$ $T3 = W[B] R[A] W[D] C3$
17	$T1 = W[D] R[B] W[B] C1$ $T2 = R[A] R[B] W[A] C2$ $T3 = R[D] W[D] C3$
18	$T1 = R[D] R[B] W[D] C1$ $T2 = W[A] R[B] W[B] C2$ $T3 = R[A] W[A] C3$
19	$T1 = R[K] R[M] W[K] C1$ $T2 = W[M] R[B] W[B] C2$ $T3 = R[M] W[M] C3$
20	$T1 = W[K] R[B] W[B] C1$ $T2 = R[M] R[B] W[M] C2$ $T3 = R[K] W[K] C3$

Таблиця 15 – Продовження

№ вар.	Приклади транзакцій
21	$T1 = R[K] \ W[K] \ C1$ $T2 = R[M] \ R[B] \ W[M] \ C2$ $T3 = W[K] \ R[B] \ W[K] \ C3$
22	$T1 = W[K] \ W[M] \ C1$ $T2 = R[M] \ R[B] \ W[M] \ C2$ $T3 = W[K] \ R[B] \ W[B] \ C3$
23	$T1 = W[K] \ W[B] \ C1$ $T2 = R[M] \ R[K] \ W[M] \ C2$ $T3 = W[K] \ R[B] \ W[M] \ C3$
24	$T1 = W[M] \ W[K] \ C1$ $T2 = R[M] \ R[K] \ W[M] \ C2$ $T3 = W[K] \ R[B] \ W[M] \ C3$
25	$T1 = W[M] \ W[K] \ C1$ $T2 = R[M] \ R[K] \ W[M] \ C2$ $T3 = W[B] \ R[K] \ W[M] \ C3$
26	$T1 = W[M] \ R[B] \ W[B] \ C1$ $T2 = R[K] \ R[B] \ W[K] \ C2$ $T3 = R[M] \ W[M] \ C3$
27	$T1 = R[M] \ R[B] \ W[M] \ C1$ $T2 = W[K] \ R[B] \ W[B] \ C2$ $T3 = R[K] \ W[K] \ C3$
28	$T1 = R[M] \ R[B] \ W[M] \ C1$ $T2 = R[K] \ W[K] \ C2$ $T3 = W[K] \ R[B] \ W[K] \ C3$
29	$T1 = W[M] \ R[B] \ W[B] \ C1$ $T2 = R[K] \ R[B] \ W[K] \ C2$ $T3 = R[M] \ W[M] \ C3$

2.2 Налаштування бази даних

Припустимо, що існує база даних (БД) зі спрощеним описом сутностей реального світу. Назва БД співпадає з іменем вашого користувача в ОС *Linux*. Приклад команди створення реляційної таблиці в БД наведено в таблиці 16 у відповідності із вашим варіантом. Також в таблиці 16 наведено приклад команди внесення одного рядка в реляційну таблицю БД. Враховуючи вищевказане, виконати наступні завдання, документуючи рішення у вигляді фрагментів знімків екранів.

2.2.1 Встановити з'єднання з БД, назва якої співпадає з іменем вашого користувача в ОС *Linux*, використовуючи користувача СКБД за таким же іменем.

- 2.2.2 Виконати команди отримання імені поточного користувача СКБД та назви БД.
- 2.2.3 У відповідності із варіантом з таблиці 16 створити реляційну таблицю.
- 2.2.4 У відповідності із варіантом з таблиці 16 додати рядок в реляційну таблицю.
- 2.2.5 Створити ще одну операцію внесення рядка в таблицю, який буде відрізнятися значеннями всіх змінних (стовпчиків) від прикладу з варіанту, а одна із змінних повинна враховувати значення із транслітерацією вашого прізвища.
- 2.2.6 Переглянути зміст таблиці, враховуючи всі стовпчики всіх рядків таблиці.

Таблиця 16 – Варіанти завдань з командами створення БД та внесення даних

№ вар.	Операція створення реляційної таблиці БД	Операції внесення даних в реляційну таблицю БД
1	<i>Create table person (p_id integer, name char(20), bd date);</i>	<i>Insert into person values (1, 'Jobs', '01/04/2000');</i>
2	<i>Create table student (s_id integer, name char(20), cours integer);</i>	<i>Insert into student values (1, 'Petrenko', 4);</i>
3	<i>Create table teacher (t_id integer, name char(20), post char(20));</i>	<i>Insert into teacher values (1, 'Gates', 'docent');</i>
4	<i>Create table department (d_id integer, name char(20), faculty char(20));</i>	<i>Insert into department values (1, 'IS', 'ICS');</i>
5	<i>Create table airplane (a_id integer, model char(20), year integer);</i>	<i>Insert into airplane values (1, 'TU-107', 1960);</i>

Таблиця 16 – Продовження

№ вар.	Операція створення реляційної таблиці БД	Операції внесення даних в реляційну таблицю БД
6	<i>Create table university (u_id integer, name char(20), year integer);</i>	<i>Insert into university values (1, 'ONPU', 1918);</i>
7	<i>Create table auto (a_id integer, name char(20), year integer);</i>	<i>Insert into auto values (1, 'BMW 5', 2003);</i>
8	<i>Create table employer (e_id integer, name char(20), salary integer);</i>	<i>Insert into employer values (1, 'Petrenko', 200);</i>
9	<i>Create table worker (w_id integer, name char(20), bday date);</i>	<i>Insert into worker values (1, 'Sidorenko', '01/04/2000');</i>
10	<i>Create table os (os_id integer, name char(20), year integer);</i>	<i>Insert into os values (1, 'Unix', 1971);</i>
11	<i>Create table planet (pl_id integer, name char(20), diameter_km integer);</i>	<i>Insert into planet values (1, 'Mars', 6 779);</i>

Таблиця 16 – Продовження

№ вар.	Операція створення реляційної таблиці БД	Операції внесення даних в реляційну таблицю БД
12	<i>Create table printer (pr_id integer, model char(20), year integer);</i>	<i>Insert into printer values (1, 'Samsung ML100', 2010);</i>
13	<i>Create table cpu (c_id integer, model char(20), year integer);</i>	<i>Insert into cpu values (1, 'Intel 8086', 1978);</i>
14	<i>Create table monitor (m_id integer, model char(20), year integer);</i>	<i>Insert into monitor values (1, 'Asus', 2020);</i>
15	<i>Create table harddisk (h_id integer, model char(20), size_tb integer);</i>	<i>Insert into harddisk values (1, 'WD', 2);</i>
16	<i>Create table memory (h_id integer, model char(20), size_gb integer);</i>	<i>Insert into memory values (1, 'Samsung DDR4', 32);</i>
17	<i>Create table file_type (ft_id integer, name char(20), small_name char(1));</i>	<i>Insert into file_type values (1, 'Directory', 'd');</i>

Таблиця 16 – Продовження

№ вар.	Операція створення реляційної таблиці БД	Операції внесення даних в реляційну таблицю БД
18	<i>Create table protocol (pr_id integer, name char(20), year integer);</i>	<i>Insert into protocol values (1, 'SSH', 1995);</i>
19	<i>Create table computer (cm_id integer, model char(20), year integer);</i>	<i>Insert into computer values (1, 'Z1', 1938);</i>
20	<i>Create table calculator (cr_id integer, model char(20), year integer);</i>	<i>Insert into calculator values (1, 'Mark I', 1944);</i>
21	<i>Create table techcompany (cc_id integer, name char(20), year integer);</i>	<i>Insert into techcompany values (1, 'Google', 1998);</i>
22	<i>Create table programlang (pl_id integer, name char(20), year integer);</i>	<i>Insert into programlang values (1, 'C', 1969);</i>
23	<i>Create table teletype (tt_id integer, model char(20), year integer);</i>	<i>Insert into teletype values (1, 'Olivetti T1', 1938);</i>

Таблиця 16 – Продовження

№ вар.	Операція створення реляційної таблиці БД	Операції внесення даних в реляційну таблицю БД
24	<i>Create table arithmometer (tt_id integer, model char(20), year integer);</i>	<i>Insert into arithmometer values (1, 'Leibniz wheel', 1673);</i>
25	<i>Create table softdrink (sd_id integer, name char(20), introduced date);</i>	<i>Insert into softdrink values (1, 'Coca-cola', '08/08/1886');</i>
26	<i>Create table schoolchild (s_id integer, name char(20), class integer);</i>	<i>Insert into schoolchild values (1, 'Fedorenko', 4);</i>
27	<i>Create table graphicscard (h_id integer, model char(20), size_gb integer);</i>	<i>Insert into graphicscard values (1, 'NVIDIA GeForce RTX 4090', 24);</i>
28	<i>Create table motherboard (h_id integer, model char(20), year integer);</i>	<i>Insert into motherboard values (1, 'AMD X670', 2020);</i>
29	<i>Create table motorbike (a_id integer, name char(20), year integer);</i>	<i>Insert into auto values (1, 'YAMAHA', 2010);</i>

2.3 Керування квазіпаралельним виконанням транзакцій з використанням команд блокування

2.3.1 Створити файл з назвою за шаблоном «*ваше прізвище_transaction_lock_1.sql*», наприклад, *blazhko_transaction_lock_1.sql*, та додати до нього операції двох транзакцій, кожна з яких повинна включати наступні операції:

- операція блокування для протоколу 1-го ступеня блокування;
- операція читання всіх стовпчиків першого рядку таблиці;
- операція зміни значення другого стовпчика таблиці у першому рядку;
- повторна операція читання всіх стовпчиків першого рядку таблиці;
- операція фіксації всіх виконаних операцій.

2.3.3 У двох псевдотерміналах виконати операції транзакцій при їх квазіпаралельному режимі роботи за умови, що одна з транзакцій стартує першою.

2.3.4 Повторити роботу транзакцій, але у першій транзакції замість операції фіксації виконати операцію відміни всіх операцій транзакції.

2.3.5 Створити файл з назвою за шаблоном «*ваше прізвище_transaction_lock_2.sql*», наприклад, *blazhko_transaction_lock_2.sql*, зі змістом файлу, створеного у пункті 2.3.1, але вже враховуючи протокол 2-го ступеня блокування.

2.3.6 Повторити роботу транзакцій з використанням протоколу 2-го ступеня блокування, але з додатковим параметром *NOWAIT*.

2.4 Керування квазіпаралельним виконанням транзакцій при наявності *Deadlock*-станів.

2.4.1 Створити файл з назвою за шаблоном «*ваше прізвище_deadlock.sql*», наприклад, *blazhko_deadlock.sql*, зі змістом файлу, створеного у пункті 2.3.1, але вже модифікованого так, щоб транзакції призводили до *Deadlock*-стану.

2.4.2 Виконати модифіковані транзакції.

Проаналізувати реакцію СКБД на операцію зміни значення стовпчика для транзакції, яка виконувалася пізніше (призвела до *Deadlock*-стану), та надати висновки за результатами аналізу з урахуванням ідентифікаторів процесів та номерів транзакцій.

2.5 Підготовка процесу *Code Review* для надання рішень завдань лабораторної роботи на перевірку викладачем

Примітка: Рішення на завдання 2.1 та 2.2 можуть бути надані під час *Online*-заняття для отримання відповідних балів. За межами *Online*-заняття виконуються всі рішення.

На веб-сервісі *GitHub* зафіксувати зміни у файлі *README.md*.

Скопіювати файли, які було створено у попередніх розділах завдань, в каталог «*Laboratory-work-10*» *Git*-репозиторію.

Оновити *Git*-репозиторій змінами нової гілки «*Laboratory-work-10*» з *GitHub*-репозиторію. Оновити *GitHub*-репозиторій змінами нової гілки «*Laboratory-work-10*» *Git*-репозиторію. Виконати запит *Pull Request*.

Примітка: Увага! Не натискайте кнопку «Merge pull request»!

Це повинен зробити лише рецензент, який є вашим викладачем!

Може бути виконано два запити *Pull Request*: під час *Online*-заняття та за межами *Online*-заняття.

Якщо запит *Pull Request* було зроблено під час *Online*-заняття, тоді рецензент-викладач перегляне рішення, надасть оцінку та закриє *Pull Request* без операції *Merge*.

Коли буде зроблено остаточний запит *Pull Request*, тоді рецензент-викладач перегляне ваше рішення та виконає злиття нової гілки та основної гілки через операцію *Merge*. Якщо рецензент знайде помилки, він повідомить про це у коментарях, які з'являться на сторінці *Pull request*.

2.6 Оцінка результатів виконання завдань лабораторної роботи

Оцінка	Умови
+2 бали	під час <i>Online</i> -заняття виконано правильні рішення завдань 2.1, 2.2 або на наступному <i>Online</i> -занятті з лабораторної роботи чи на консультації отримано правильну відповідь на два запитання, які стосуються виконаних завдань
+2 бали	1) всі рішення роботи відповідають завданням 2) <i>Pull Request</i> представлено не пізніше найближчої консультації після офіційного заняття із захисту лабораторної роботи
-0.5 балів за кожну помилку	в рішенні є помилка, про яку вказано в <i>Code Review</i>
-1 бал	<i>Pull Request</i> представлено пізніше часу завершення найближчої консультації після офіційного заняття із захисту лабораторної роботи за кожний тиждень запізнення
+2 бали	на наступному <i>Online</i> -занятті з лабораторної роботи або на консультації отримано правильну відповідь на два запитання, які стосуються виконаних завдань

Література

1. Блажко О.А. Відео-запис лекції «Керування процесами-транзакціями в базах даних». URL: <https://youtu.be/oickt-zij24>

Контрольні запитання

1. Що таке квазіпаралельна успішна та частково-успішна історія виконання транзакцій?
2. Дайте визначення поняттю "атомарність" з *ACID*-вимог.
3. Дайте визначення поняттю "узгодженість" з *ACID*-вимог.
4. Опишіть проблеми феномену "Брудне Читання".
5. Опишіть проблеми феномену "Брудна Модифікація".
6. Опишіть проблеми феномену "Неповторне Читання".
7. Які переваги використання СКБД перед звичайною роботою із файлами ?
8. Як працює протокол 1-го ступеня блокування?
9. Чим відрізняється протокол 2-го ступеня блокування від протоколу 1-го ступеня блокування?
10. Який феномен виключає протокол 1-го ступеня блокування, а який феномен виключає протокол 2-го ступеня блокування?
11. В яких станах може перебувати транзакція? При яких умовах та завдяки яким операціям вона може перейти до цих станів?
12. Опишіть структуру матриці сумісності блокувань?
13. Якими командами встановлюються типи блокувань у СКБД *PostgreSQL*?
14. Як вказати СКБД про відмову транзакції чекати на завершення роботи іншої транзакції при несумісності блокувань? Яка буде реакція СКБД на несумісність блокувань?
15. Що таке *Deadlock*-стан при роботі транзакцій?
16. Наведіть приклади історій квазіпаралельного виконання транзакцій, які можуть призвести до *Deadlock*-стану.
17. Як можна знайти *Deadlock*-стан?
18. Як СКБД *PostgreSQL* вирішує проблему *Deadlock*-станів транзакцій?
19. Яка особливість взаємовідносин *psql*-процесу та СКБД-процесу *PostgreSQL*?