A4 + A5

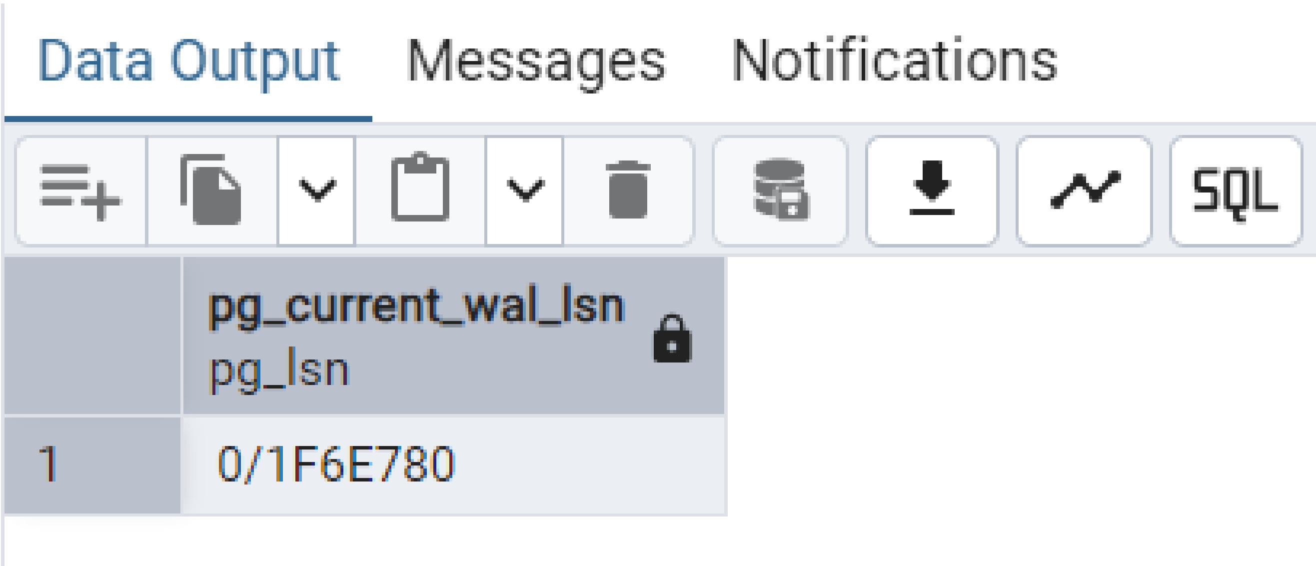
Type	Homework

Write-Ahead-Log (WAL)

A file where insert, update, delete information is written. it is fail safe.

1. Return the current LSN of your WAL file

```
SELECT pg_current_wal_lsn();
```



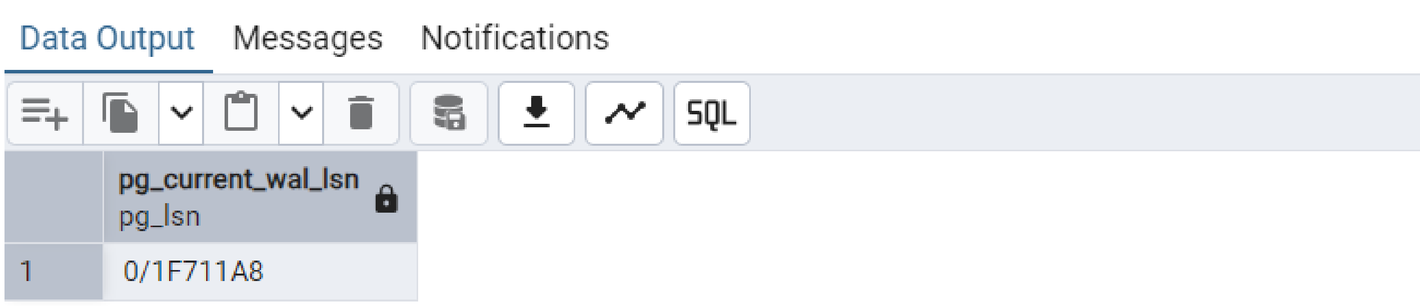
	pg_current_wal_lsn pg_lsn
1	0/1F6E780

`0/` is WAL segment, `1F6E780` is offset within the segment.

2. Run the transaction on next slide. (Make sure that the Pks do NOT yet exist in your table lesson.)

```
Begin;
update student set s_balance = s_balance - 3 where s_username= 'Rose';
update teacher set t_payment = t_payment + 3 where t_id = 1;
select t_payment from teacher where t_id = 1;
INSERT INTO lesson(t_id, lesson_time, s_username, subjectcode)
VALUES (1, '2024-03-03 05:22:12.000000', 'Rose', 'EN');
INSERT INTO lesson(t_id, lesson_time, s_username, subjectcode)
VALUES (1, '2024-03-04 05:22:12.000000', 'Rose', 'EN');
INSERT INTO lesson(t_id, lesson_time, s_username, subjectcode)
VALUES (1, '2024-03-05 05:22:12.000000', 'Rose', 'EN');
Commit;
```

3. Return the current LSN of your WAL file



	pg_current_wal_lsn pg_lsn
1	0/1F711A8

4. Dump the WAL from the first to the second LSN

```
pg_waldump -s 0/1F6E780 -e 0/1F711A8 -p "D:/postgreSQL/17/data/pg_wal"
```

```
PS C:\Users\barba> pg_waldump -s 0/1F6E780 -e 0/1F711A8 -p "D:/postgreSQL/17/data/pg_wal"
rmgr: Heap        len (rec/tot):     65/   853, tx:        805, lsn: 0/01F6E780, prev 0/01F6E748, desc: HOT_UPDATE old_xmax: 805, old_off: 7, old_infobits:
[], flags: 0x60, new_xmax: 0, new_off: 9, blkref #0: rel 1663/16478/16493 blk 0 FPW
rmgr: Heap        len (rec/tot):     65/  6461, tx:        805, lsn: 0/01F6EAD8, prev 0/01F6E780, desc: HOT_UPDATE old_xmax: 805, old_off: 20, old_infobits:
 [], flags: 0x00, new_xmax: 0, new_off: 21, blkref #0: rel 1663/16478/16499 blk 0 FPW
rmgr: Heap        len (rec/tot):     54/   830, tx:        805, lsn: 0/01F70430, prev 0/01F6EAD8, desc: INSERT off: 12, flags: 0x00, blkref #0: rel 1663/164
78/16482 blk 0 FPW
rmgr: Standby     len (rec/tot):     54/    54, tx:          0, lsn: 0/01F70770, prev 0/01F70430, desc: RUNNING_XACTS nextXid 806 latestCompletedXid 804 old
estRunningXid 805; 1 xacts: 805
rmgr: Btree       len (rec/tot):     53/   429, tx:        805, lsn: 0/01F707A8, prev 0/01F70770, desc: INSERT_LEAF off: 4, blkref #0: rel 1663/16478/16517
blk 1 FPW
rmgr: Btree       len (rec/tot):     53/   429, tx:        805, lsn: 0/01F70958, prev 0/01F707A8, desc: INSERT_LEAF off: 8, blkref #0: rel 1663/16478/16533
blk 1 FPW
rmgr: Heap        len (rec/tot):     54/    54, tx:        805, lsn: 0/01F70B08, prev 0/01F70958, desc: LOCK xmax: 805, off: 21, infobits: [LOCK_ONLY, KEYSH
R_LOCK], flags: 0x00, blkref #0: rel 1663/16478/16499 blk 0
rmgr: Heap        len (rec/tot):     59/  1091, tx:        805, lsn: 0/01F70B40, prev 0/01F70B08, desc: LOCK xmax: 805, off: 1, infobits: [LOCK_ONLY, KEYSHR
_LOCK], flags: 0x00, blkref #0: rel 1663/16478/16505 blk 0 FPW
rmgr: Heap        len (rec/tot):     54/    54, tx:        805, lsn: 0/01F70F88, prev 0/01F70B40, desc: LOCK xmax: 805, off: 9, infobits: [LOCK_ONLY, KEYSHR
_LOCK], flags: 0x00, blkref #0: rel 1663/16478/16493 blk 0
rmgr: Heap        len (rec/tot):     79/    79, tx:        805, lsn: 0/01F70FC0, prev 0/01F70F88, desc: INSERT off: 13, flags: 0x00, blkref #0: rel 1663/164
78/16482 blk 0
rmgr: Btree       len (rec/tot):     72/    72, tx:        805, lsn: 0/01F71010, prev 0/01F70FC0, desc: INSERT_LEAF off: 5, blkref #0: rel 1663/16478/16517
blk 1
rmgr: Btree       len (rec/tot):     72/    72, tx:        805, lsn: 0/01F71058, prev 0/01F71010, desc: INSERT_LEAF off: 9, blkref #0: rel 1663/16478/16533
blk 1
rmgr: Heap        len (rec/tot):     79/    79, tx:        805, lsn: 0/01F710A0, prev 0/01F71058, desc: INSERT off: 14, flags: 0x00, blkref #0: rel 1663/164
78/16482 blk 0
rmgr: Btree       len (rec/tot):     72/    72, tx:        805, lsn: 0/01F710F0, prev 0/01F710A0, desc: INSERT_LEAF off: 6, blkref #0: rel 1663/16478/16517
blk 1
rmgr: Btree       len (rec/tot):     72/    72, tx:        805, lsn: 0/01F71138, prev 0/01F710F0, desc: INSERT_LEAF off: 10, blkref #0: rel 1663/16478/16533
 blk 1
rmgr: Transaction len (rec/tot):     34/    34, tx:        805, lsn: 0/01F71180, prev 0/01F71138, desc: COMMIT 2025-03-26 03:10:29.937503 Georgian Standard
Time
```

1. Hot Updates
    - `old_xmax: 805` - This is **transaction ID** that invalidated the **old row**.
    - `old_off: 20` - This represents **offset** of the old row within the page.
    - `new_xmax: 0` - This is **transaction ID** that (will or not) delete the new row.
    - `new_off: 21` - This is the **offset** of the newly inserted row within the same page.
2. why inserts into 4 different btrees?

- `INSERT` - Insert new row in **Heap**.
- `INSERT_LEAF` - Insert new index in **BTree**.

BTree data structures are added to **primary key** and **unique constraints**.

In our case tuple, `t_id` and `lesson_time`, form primary composite key (1 **BTree**).

We also have unique constraint on tuple, `t_id` and `s_username`, forming second **BTree**.

3. why inserts into the btree leaves?

Since the **actual index data is stored in leaf nodes**, PostgreSQL only logs `INSERT_LEAF`.

Dirty Read Anomaly in PostgreSQL?

reads data written by a concurrent uncommitted transaction.

Transaction 1

```
Begin;
drop table if exists transaction_log;
CREATE TEMPORARY TABLE transaction_log (message_text varchar(50), t_payment_value INT);
INSERT INTO transaction_log (message_text, t_payment_value) select 'Payment_amount',t_payment FROM teacher WHER
UPDATE teacher SET t_payment = 0 WHERE t_id = 1;

SELECT pg_sleep(30);

INSERT INTO transaction_log (message_text, t_payment_value) select 'after_reset_balance',t_payment FROM teacher
WHERE t_id = 1;
rollback;
```

Transaction 2

```
Begin;
drop table if exists transaction_log;
CREATE TEMPORARY TABLE transaction_log (t_payment_value INT);
```

```
INSERT INTO transaction_log (t_payment_value) select t_payment FROM teacher WHERE t_id = 1;
COMMIT;
```

Consistent result would be:

`t_payment=0` shouldn't have been for `t_id=1` it should have had its original value.

Inconsistent result ("dirty read") would be:

it reads uncommited data which is t_payment_value = 0, thus causing dirty read.

```
agency=# SELECT * FROM transaction_log;
 t_payment_value
-----------------
               0
(1 row)
```

Does it make any difference for the read in session 2 if the transaction in session 1 commits or rollbacks?

if it would've rolledback it would've had its original value and for the commit it would have the value of 0.

Non-Repeatable Read Anomaly

reads the same row twice but gets different data each time. in repeatable-read and serializable in postgreSQL.

Transaction 1

```
BEGIN TRANSACTION ISOLATION LEVEL REPEATABLE READ;
SELECT s_username FROM student WHERE s_balance  < 5;
SELECT pg_sleep(20);
SELECT s_username FROM student WHERE s_balance  < 5;
COMMIT;
```

```
agency=*# SELECT s_username FROM student WHERE s_balance  < 5;
 s_username
------------
 Witzi
 Wolf
 Darthvader
(3 rows)


agency=*# COMMIT;
COMMIT
agency=# BEGIN TRANSACTION ISOLATION LEVEL REPEATABLE READ;
BEGIN
agency=*# SELECT s_username FROM student WHERE s_balance  < 5;
 s_username
------------
 Witzi
 Wolf
 Darthvader
(3 rows)
```

Transaction 2

```
BEGIN TRANSACTION ISOLATION LEVEL REPEATABLE READ;
UPDATE student SET s_balance = s_balance + 5 WHERE s_username IN
(SELECT s_username FROM student WHERE s_balance < 5 LIMIT 1);
COMMIT;
```

```
BEGIN
agency=*# UPDATE student SET s_balance = s_balance + 5 WHERE s_username IN
agency-*# (SELECT s_username FROM student WHERE s_balance < 5 LIMIT 1);
UPDATE 1
agency=*# COMMIT;
COMMIT
agency=# select s_username from student where s_balance < 5;
 s_username
------------
 Darthvader
(1 row)
```

Phantom Read Anomaly in PostgreSQL?

re-executes a query and sees new rows inserted by another transaction, phantom read in postgreSQL in repeatable read and serializable. In other only serializable.

Transaction 1

```
BEGIN TRANSACTION ISOLATION LEVEL REPEATABLE READ;
SELECT count(*) FROM teacher WHERE t_payment <= 5;
SELECT pg_sleep(20);
SELECT count(*) FROM teacher WHERE t_payment <= 5;
COMMIT;
```

```
agency=# BEGIN TRANSACTION ISOLATION LEVEL REPEATABLE READ;
BEGIN
agency=*# SELECT count(*) FROM teacher WHERE t_payment <= 5;
 count
-------
    18
(1 row)


agency=*# SELECT pg_sleep(20);
 pg_sleep
----------

(1 row)


agency=*# SELECT count(*) FROM teacher WHERE t_payment <= 5;
 count
-------
    18
(1 row)


agency=*# COMMIT;
COMMIT
```

Transaction 2

```
BEGIN;
INSERT INTO public.teacher (t_id, t_name, t_mail, t_postalcode, t_dob, t_gender, t_education, t_remark, t_payment)
VALUES (25, 'lamara', 'lamar@gmail.com', 8000, '1880-10-10', 'f', 'Bachelor', NULL, 0);
COMMIT;
```

```
agency=# SELECT count(*) FROM teacher WHERE t_payment <= 5;
 count
-------
    19
(1 row)
```

2PL

Given is the following schedule. How would it work under 2PL?

```
T1:Read(A) T2:Read(A) T1:Write(A) T2:Write(A) T1:Read(B) T2:Read(B) T1:Write(B)
T2:Write(B)
```

What is the result?

Operation	T1	T2
1	BEGIN	
2	S-lock on A	BEGIN
3		S-lock on A
4	can’t put E-lock on A in T1, because of having to wait for T2	
5		can’t put E-lock on A in T2, because of having to wait for T1
6	Deadlock	Deadlock

Deadlock

1. Write the code for T1, T2 and T3 and let them run parallel in 3 sessions.

Transaction 1

```
BEGIN;
UPDATE student SET s_balance = s_balance - 3 WHERE s_username = 'Mickey';
SELECT pg_sleep(40);
UPDATE teacher SET t_payment = t_payment + 3 WHERE t_id = 1;
COMMIT;
```

Transaction 2

```
BEGIN;
UPDATE teacher SET t_payment = t_payment + 3 WHERE t_id = 1;
SELECT pg_sleep(35);
UPDATE student SET s_balance = s_balance - 3 WHERE s_username = 'Rose';
COMMIT;
```

Transaction 3

```
BEGIN;
UPDATE student SET s_balance = s_balance - 3 WHERE s_username = 'Rose';
SELECT pg_sleep(30);
UPDATE student SET s_balance = s_balance - 3 WHERE s_username = 'Mickey';
UPDATE teacher SET t_payment = t_payment + 6 WHERE t_id = 2;
COMMIT;
```

2. Which transaction is rolled back, which commit in what sequence?

First Transaction 2 rollbacked

```
ERROR:  deadlock detected
DETAIL:  Process 11772 waits for ShareLock on transaction 822; blocked by process 10072.
Process 10072 waits for ShareLock on transaction 820; blocked by process 7184.
Process 7184 waits for ShareLock on transaction 821; blocked by process 11772.
HINT:  See server log for query details.
CONTEXT:  while updating tuple (0,9) in relation "student"
agency=!# COMMIT;
ROLLBACK
```

Second Transaction 1 committed

```
agency=*# SELECT pg_sleep(40);
 pg_sleep
----------

(1 row)


agency=*# UPDATE teacher SET t_payment = t_payment + 3 WHERE t_id = 1;
UPDATE 1
agency=*# COMMIT;
COMMIT
```

Third Transaction 3 committed

```
agency=*# UPDATE student SET s_balance = s_balance - 3 WHERE s_username = 'Rose';
UPDATE 1
agency=*# SELECT pg_sleep(30);
 pg_sleep
----------

(1 row)


agency=*# UPDATE student SET s_balance = s_balance - 3 WHERE s_username = 'Mickey';
UPDATE 1
agency=*# UPDATE teacher SET t_payment = t_payment + 6 WHERE t_id = 2;
UPDATE 1
agency=*# COMMIT;
COMMIT
```

3. What is the error message?

```
ERROR:  deadlock detected
DETAIL:  Process 11772 waits for ShareLock on transaction 822; blocked by process 10072.
Process 10072 waits for ShareLock on transaction 820; blocked by process 7184.
Process 7184 waits for ShareLock on transaction 821; blocked by process 11772.
HINT:  See server log for query details.
CONTEXT:  while updating tuple (0,9) in relation "student"
agency=!# COMMIT;
ROLLBACK
```

deadlock has been detected and second transaction has been aborted.

4. Explain the problem and the result.

the problem is that three transactions that run cause a deadlock, thus making one of the transactions abort and others to commit.

Waits-For-Graph

```
graph LR
   10072 --- 7184
   7184 --- 1172
     1172 --- 10072
```

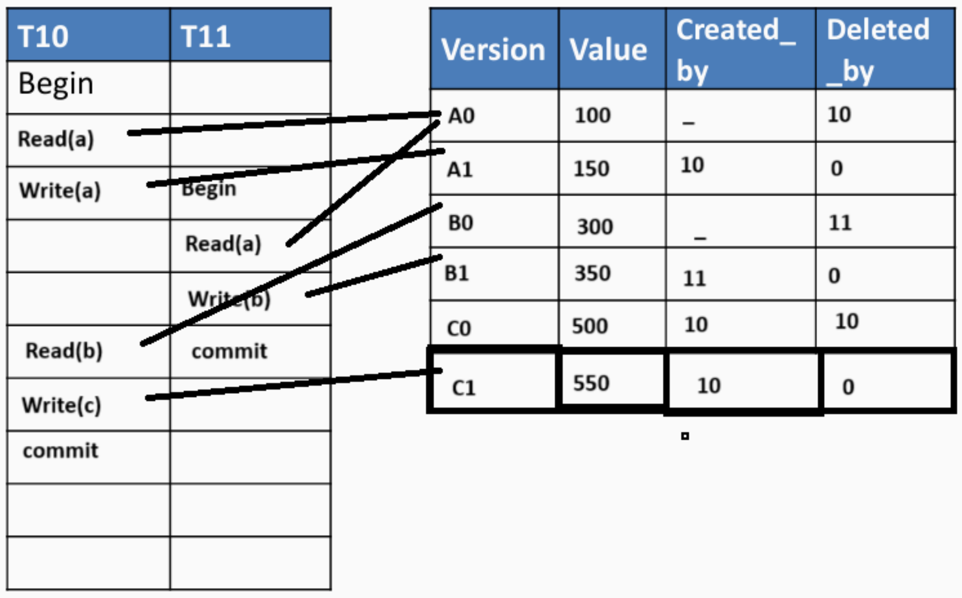
second transaction has caused the error it is the victim, it is the transaction that triggers the circuit, this happens because of pg_sleep, if it was ran separately as blocks T3 would've been aborted.

MVCC

```
 T1 read(a) T1write(a) T2read(a) T2write(b) T1 read(b) T1write(c)
```

Assume that all writes are updates. Objects a, b and c already exist.
How does the execution under MVCC look like? Do the two transactions run through?

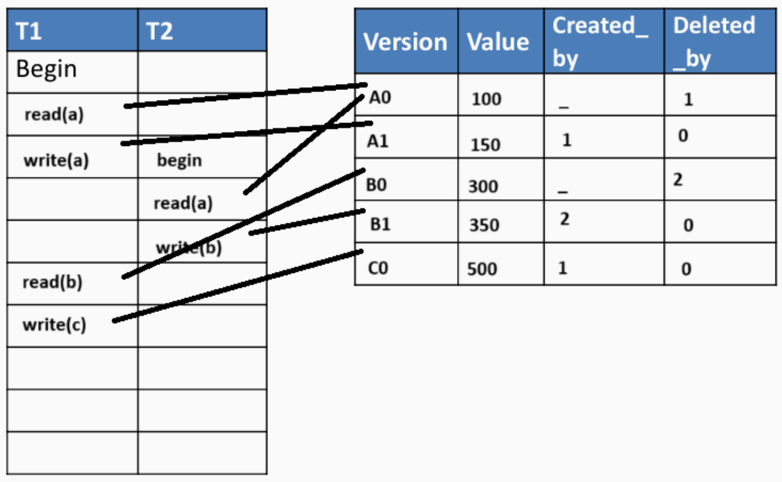
Given is the following schedule s:
T1 read(a) T1write(a) T2read(a) T2write(b) T1 read(b) T1write(c)
Assume that all writes are updates. Objects a,b,a nd c already exist.
How does the execution under MVCC look like? Do the two transactiobn run through?



T10	T11
Begin	
Read(a)	
Write(a)	Begin
	Read(a)
	Write(b)
Read(b)	commit
Write(c)	
commit	

Version	Value	Created_by	Deleted_by
A0	100	_	10
A1	150	10	0
B0	300	_	11
B1	350	11	0
C0	500	10	10
C1	550	10	0

Given is the following schedule s:
T1 read(a) T1 write(a) T2 read(a) T2 write(b) T1 read(b) T1 write (c)
Assume that object a and b exist. T2 write(b) is a delete and T1 write (c) is an insert. How does the execution under MVCC look like?



T1	T2
Begin	
read(a)	
write(a)	begin
	read(a)
	write(b)
read(b)	
write(c)	

Version	Value	Created_by	Deleted_by
A0	100	_	1
A1	150	1	0
B0	300	_	2
B1	350	2	0
C0	500	1	0

13
eva.knirsch@kiu.edu.ge

Write Skew anomaly

Creating table

```
CREATE TABLE personnel (
   name VARCHAR(200) PRIMARY KEY,
   on_duty BOOLEAN
);
```

```
INSERT INTO personnel (name, on_duty) VALUES
('Vanhelsing', TRUE),
('Bobby', TRUE),
('Charlieputh', TRUE),
('Davidthethird', FALSE),
('Eve', FALSE),
('Alice', TRUE),
('Bob', TRUE);
```

Transaction 1

```
BEGIN TRANSACTION ISOLATION LEVEL REPEATABLE READ;
DO $$
DECLARE n_on_duty INTEGER;
BEGIN

    SELECT count(*) INTO n_on_duty FROM personnel WHERE on_duty = TRUE;

    IF n_on_duty > 2 THEN
        UPDATE personnel SET on_duty = FALSE WHERE name = 'Alice';
    END IF;
END $$;
SELECT pg_sleep(20);
COMMIT;
```

```
agency=# BEGIN TRANSACTION ISOLATION LEVEL REPEATABLE READ;
BEGIN
agency=*# DO $$
agency$*# DECLARE n_on_duty INTEGER;
agency$*# BEGIN
agency$*#
agency$*#     SELECT count(*) INTO n_on_duty FROM personnel WHERE on_duty = TRUE;
agency$*#
agency$*#     IF n_on_duty > 2 THEN
agency$*#         UPDATE personnel SET on_duty = FALSE WHERE name = 'Alice';
agency$*#     END IF;
agency$*# END $$;
DO
agency=*# SELECT pg_sleep(20);
 pg_sleep
----------

(1 row)


agency=*# COMMIT;
```

Transaction 2

```
BEGIN TRANSACTION ISOLATION LEVEL REPEATABLE READ;
DO $$
DECLARE n_on_duty INTEGER;
BEGIN
    SELECT count(*) INTO n_on_duty FROM personnel WHERE on_duty = TRUE;

    IF n_on_duty > 2 THEN
    UPDATE personnel SET on_duty = FALSE WHERE name = 'Bob';
    END IF;
END $$;
SELECT pg_sleep(20);
COMMIT;
```

```
agency=# BEGIN TRANSACTION ISOLATION LEVEL REPEATABLE READ;
BEGIN
agency=*# DO $$
agency$*# DECLARE n_on_duty INTEGER;
agency$*# BEGIN
agency$*#     SELECT count(*) INTO n_on_duty FROM personnel WHERE on_duty = TRUE;
agency$*#
agency$*#     IF n_on_duty > 2 THEN
agency$*#       UPDATE personnel SET on_duty = FALSE WHERE name = 'Bob';
agency$*#     END IF;
agency$*# END $$;
DO
agency=*# SELECT pg_sleep(20);
 pg_sleep
----------

(1 row)


agency=*# COMMIT;
COMMIT
```

```
agency=#     SELECT count(*) INTO n_on_duty FROM personnel WHERE on_duty = TRUE;
SELECT 1
```

Running this queries in repeatable read won't change anything since `n_on_duty` is already set and this isolation level treats both `update` actions valid. we can only try `SERIALIZABLE`.

Delete from personnel

```
DELETE FROM personnel;
INSERT INTO personnel (name, on_duty) VALUES
('Vanhelsing', TRUE),
('Bobby', TRUE),
('Charlieputh', TRUE),
('Davidthethird', FALSE),
('Eve', FALSE),
('Alice', TRUE),
('Bob', TRUE);
```

replace `repeatable read` with `serializable`.

Second transaction throws an error.

```
agency=*# COMMIT;
ERROR:  could not serialize access due to read/write dependencies among transactions
DETAIL:  Reason code: Canceled on identification as a pivot, during commit attempt.
HINT:  The transaction might succeed if retried.
agency=#
```

second transaction is not committed and `on_duty` doctors remain at least 4.

```
agency=# SELECT count(*) FROM personnel WHERE on_duty = 'True';
 count
-------
     4
(1 row)
```