

資料庫管理 HW04

B12508026 戴偉璿

November 17, 2025

1. To check if PostgreSQL can avoid dirty read, I design two transactions:

- Transaction A: Update balance to 999 of account_id 1

```
1      begin;
2      update accounts set balance = 999 where account_id = 1;
3      commit;
```

- Transaction B: Read the record.

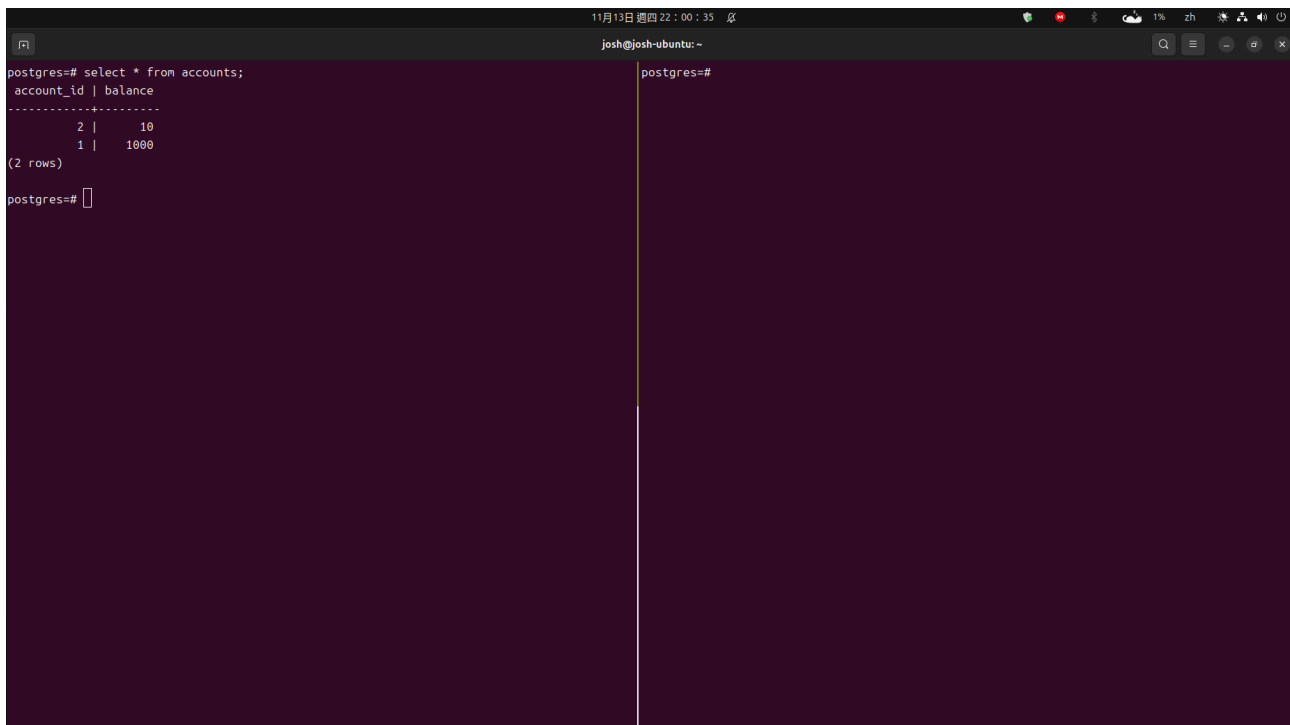
```
1      begin; select * from accounts where account_id = 1; commit;
2
3
```

The execution steps are as follows:

- (a) Transaction A begins.
- (b) Transaction A updates balance to 999 of account_id 1, but does not commit yet.
- (c) Transaction B begins.
- (d) Transaction B reads the record of account_id 1.
- (e) Transaction B gets the old balance (not 999), which means dirty read is avoided.
- (f) Transaction B commits.
- (g) Transaction A commits.
- (h) Transaction B begins.
- (i) Transaction B reads the record of account_id 1.
- (j) Transaction B gets the new balance (999) after Transaction A commits.
- (k) Transaction B commits.

Following are the screenshots of each step. Left panel shows Transaction A, right panel shows Transaction B. Figure 1 shows the original status of the accounts table, we can see the balance of account_id 1 is 1000. Figure 2 shows Transaction A updates balance to

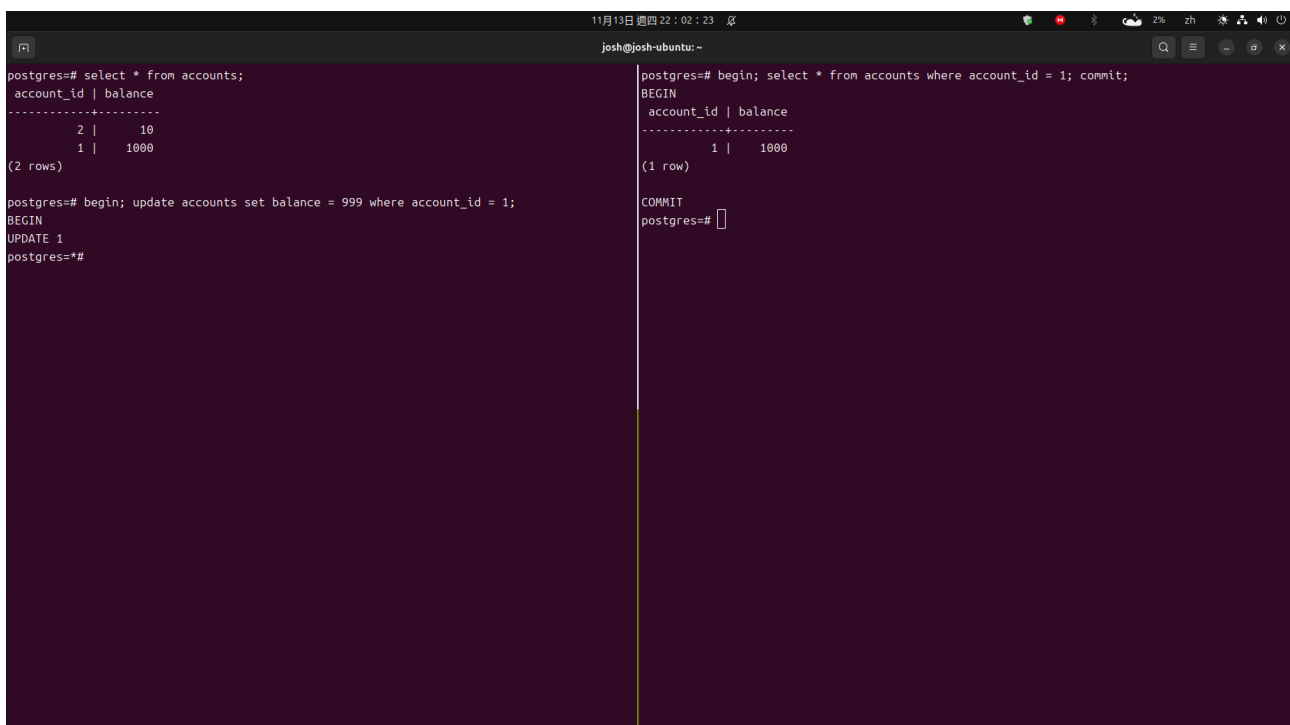
999 of account_id 1 but does not commit yet, so that Transaction B still reads the old balance (1000). Figure 3 shows Transaction A commits, and then Transaction B reads the new balance (999) of account_id 1. You can determine the execution order by the system time shown in the top of each figure.



```
11月13日 週四 22:00:35
josh@josh-ubuntu: ~
postgres=# select * from accounts;
 account_id | balance 
-----
          2 |      10
          1 |    1000
(2 rows)

postgres=#
```

Figure 1: Original status of the accounts table



```
11月13日 週四 22:02:23
josh@josh-ubuntu: ~
postgres=# select * from accounts;
 account_id | balance 
-----
          2 |      10
          1 |    1000
(2 rows)

postgres=# begin; update accounts set balance = 999 where account_id = 1;
BEGIN
UPDATE 1
postgres=#

postgres=# begin; select * from accounts where account_id = 1; commit;
BEGIN
 account_id | balance 
-----
          1 |    1000
(1 row)

COMMIT
postgres=#
```

Figure 2: Transaction A updates balance to 999 of account_id 1, but does not commit yet

```

josh@josh-ubuntu: ~
11月13日 週四 22:02:45 京

postgres=# select * from accounts;
 account_id | balance 
-----
 2 | 10
 1 | 1000
(2 rows)

postgres=# begin; update accounts set balance = 999 where account_id = 1;
BEGIN
UPDATE 1
postgres=# commit;
COMMIT
postgres=#

postgres=# begin; select * from accounts where account_id = 1; commit;
BEGIN
 account_id | balance 
-----
 1 | 1000
(1 row)

COMMIT
postgres=# begin; select * from accounts where account_id = 1; commit;
BEGIN
 account_id | balance 
-----
 1 | 999
(1 row)

COMMIT
postgres=#

```

Figure 3: Transaction A commits, Transaction B reads the new balance (999) of account_id 1

With the experiments above, we can see that PostgreSQL can avoid dirty read.

2. (a) A conflict occurs when two transactions access the same data item and at least one of the accesses is a write operation. For item X , three conflicts occurs between $\{O_{11}, O_{23}\}$, $\{O_{17}, O_{21}\}$, $\{O_{17}, O_{23}\}$; for item Y , there are no conflicts; for item Z , two conflicts occurs between $\{O_{15}, O_{24}\}$, $\{O_{15}, O_{26}\}$.

- (b) The serial schedule of $T_2 \rightarrow T_1$ is as follows:

$$O_{21} \rightarrow O_{23} \rightarrow O_{24} \rightarrow O_{26} \rightarrow O_{11} \rightarrow O_{12} \rightarrow O_{13} \rightarrow O_{15} \rightarrow O_{17}.$$

To analyze the conflicting operation pairs, we can discuss by data items:

For item X , the operation order must obey: $O_{21} \prec O_{23} \prec O_{17}$, $O_{23} \prec O_{11} \prec O_{17}$

For item Y , there are no conflicts, so there is no constraint.

For item Z , the operation order must obey: $O_{24} \prec O_{26} \prec O_{15}$

Therefore, one such conflict-equivalent non-serial schedule is:

$$O_{21} \rightarrow O_{23} \rightarrow O_{11} \rightarrow O_{24} \rightarrow O_{26} \rightarrow O_{12} \rightarrow O_{13} \rightarrow O_{15} \rightarrow O_{17}.$$

- (c) The serial schedule of $T_1 \rightarrow T_2$ is as follows:

$$O_{11} \rightarrow O_{12} \rightarrow O_{13} \rightarrow O_{15} \rightarrow O_{17} \rightarrow O_{21} \rightarrow O_{23} \rightarrow O_{24} \rightarrow O_{26}.$$

Consider the last operation of T_1 and the first operation of T_2 , we have $O_{17} \prec O_{21}$ because O_{17} is the write operation and they access the same data item X .

To maintain the order within the transactions, O_{17} must be the last operation in the schedule and O_{21} must be the first operation in the schedule. From the analyze above, O_{21} must after O_{17} . Therefore, there is no such conflict-equivalent non-serial schedule.

3. Lost update often occurs when two transactions read the same version of a data item and then update it based on that old value. This causes one update to overwrite the other, resulting in one update being lost. To address this issue, we can acquire a read lock before reading the data item so that other transactions cannot write to it until the read lock is released. This prevents the read operation from being interfered with by concurrent writes. Before updating the data, we can upgrade the read lock to a write lock to prevent other transactions from reading or writing the item until the write lock is released. This prevents other transactions from accessing the data while it is being updated, ensuring data integrity.
4. (a) This SQL statement queries the top 10 reserved sales departure from station id 1030 and arrival station id 1000 at 2023-08-01.