資料庫管理 HW04

B12508026 戴偉璿

November 18, 2025

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

   - Transaction A: Update balance to 999 of account_id 1

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

   - Transaction B: Read the record.

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

   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.

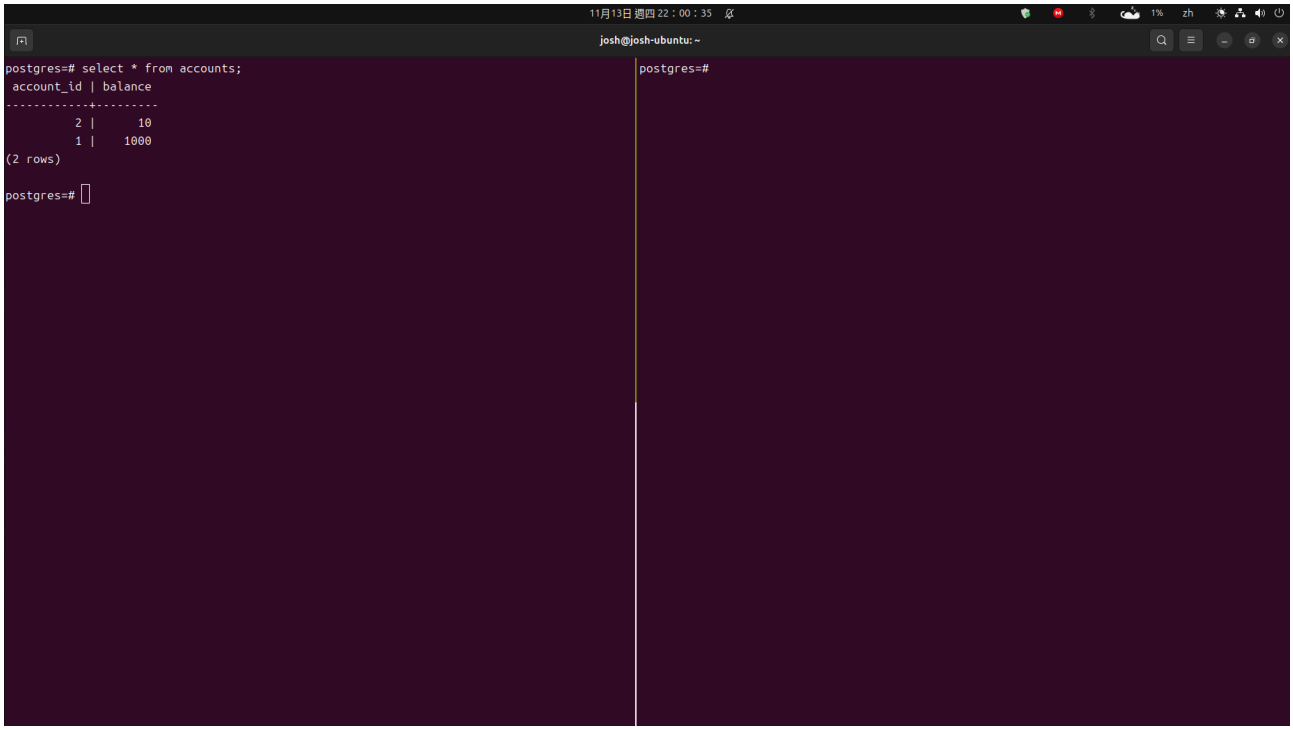

Figure 1: Orginal status of the accounts table

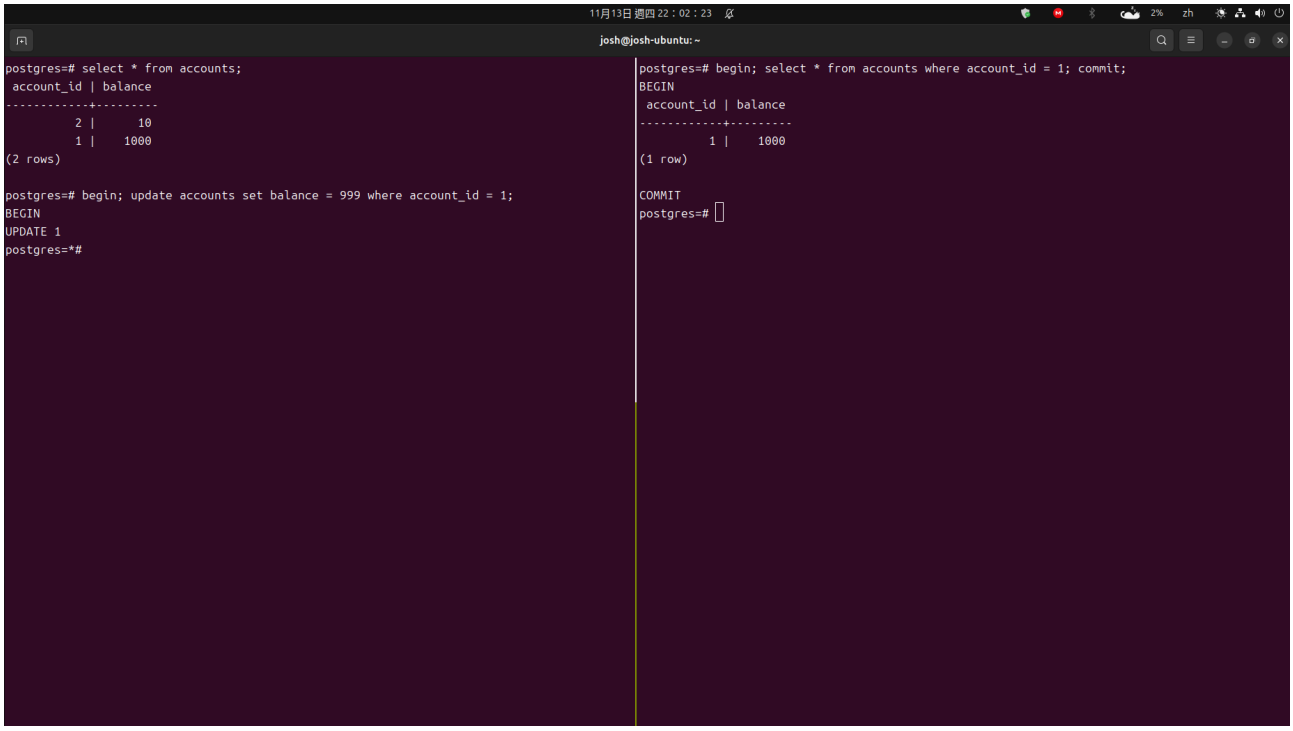

Figure 2: Transaction A updates balance to 999 of account_id 1, but does not commit yet

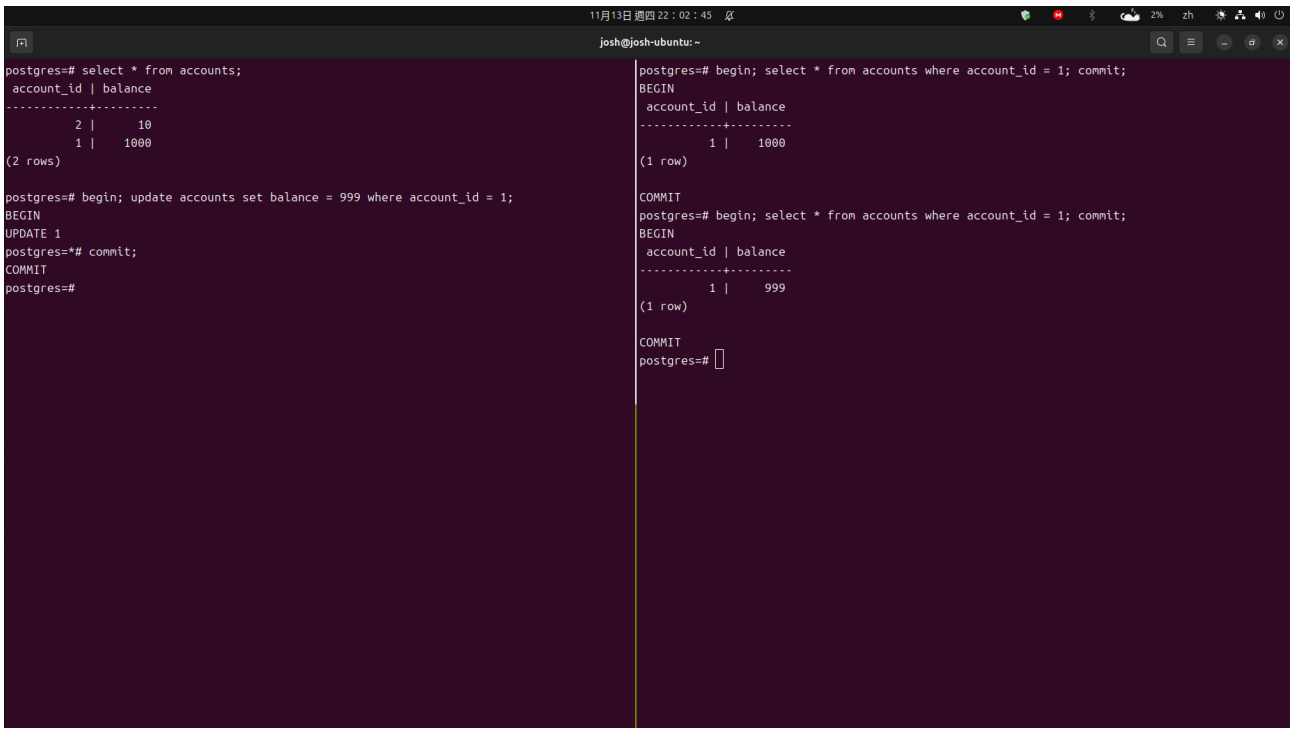

Figure 3: Transaction A commits, Transaction B reads the new balance (999) of account_id 1

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 acceses is a write operation. For item X, three conflicts occurs between $\{O_{11}, O_{23}\}$, $\{O_{17}, O_{21}\}$, $\{O_{17}, O_{23}\}$; for itme 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 $T_2 \to T_1$ is as follows:

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

   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:

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

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

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

   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 for trips that depart from station_id 1030 and arrive at station_id 1000 during the period from 2023-08-01 to 2023-08-31, and the train must depart after 06:00. It joins the PASS table with itself on trip_id, where p1 represents the departure station (1030) and p2 represents the arrival station (1000). After joining, it selects each trip's departure and arrival time, and uses a subquery to count how many tickets were reserved for that same trip and same station pair within the specified date range. Finally, the results are ordered by departure time and limited to the first 10 row

   (b) Following is hte query plan generated by PostgreSQL for the SQL statement above, the estimated cost is 560.17.

```
Limit  (cost=66.83..196.66 rows=10 width=28)
  ->  Result  (cost=66.83..560.17 rows=38 width=28)
        ->  Sort  (cost=66.83..66.93 rows=38 width=20)
              Sort Key: p1.depart_time
              ->  Hash Join  (cost=35.44..66.01 rows=38 width=20)
                    Hash Cond: (p2.trip_id = p1.trip_id)
                    Join Filter: (p1.depart_time < p2.arrive_time)
                    ->  Seq Scan on pass p2  (cost=0.00..30.06 rows=195 width=12)
                          Filter: (station_id = 1000)
                    ->  Hash  (cost=34.08..34.08 rows=109 width=12)
                          ->  Seq Scan on pass p1  (cost=0.00..34.08 rows=109 width=12)
                                Filter: ((depart_time > '06:00:00'::time without time zone) AND
                                ↪  (station_id = 1030))
        SubPlan 1
          ->  Aggregate  (cost=12.96..12.97 rows=1 width=8)
                ->  Index Only Scan using idx_trip_id_on_reserved_ticket on reserved_ticket
                ↪  (cost=0.43..12.58 rows=151 width=0)
                      Index Cond: ((trip_id = p1.trip_id) AND (arrive_station_id = 1000) AND
                      ↪  (depart_station_id = 1030) AND (travel_date >= '2023-08-01'::date)
                      ↪  AND (travel_date <= '2023-08-31'::date))

```

(c) Following is the code to create index on `trip_id`, `depart_station_id`, and `arrive_station_id`:

```
create index idx_trip_id on pass(trip_id);

create index idx_depart_station_id on reserved_ticket(depart_station_id);

create index idx_arrive_station_id on reserved_ticket(arrive_station_id);
```

Following is the query plan after creating index on `trip_id`

```
 Limit  (cost=66.83..196.66 rows=10 width=28)
   ->  Result  (cost=66.83..560.17 rows=38 width=28)
         ->  Sort  (cost=66.83..66.93 rows=38 width=20)
               Sort Key: p1.depart_time
               ->  Hash Join  (cost=35.44..66.01 rows=38 width=20)
                     Hash Cond: (p2.trip_id = p1.trip_id)
                     Join Filter: (p1.depart_time < p2.arrive_time)
                     ->  Seq Scan on pass p2  (cost=0.00..30.06 rows=195 width=12)
                           Filter: (station_id = 1000)
                     ->  Hash  (cost=34.08..34.08 rows=109 width=12)
                           ->  Seq Scan on pass p1  (cost=0.00..34.08 rows=109 width=12)
                                 Filter: ((depart_time > '06:00:00'::time without time zone) AND
                                 ↪  (station_id = 1030))
         SubPlan 1
           ->  Aggregate  (cost=12.96..12.97 rows=1 width=8)
                 ->  Index Only Scan using idx_trip_id_on_reserved_ticket on reserved_ticket
                 ↪  (cost=0.43..12.58 rows=151 width=0)
                       Index Cond: ((trip_id = p1.trip_id) AND (arrive_station_id = 1000) AND
                       ↪  (depart_station_id = 1030) AND (travel_date >= '2023-08-01'::date)
                       ↪  AND (travel_date <= '2023-08-31'::date))
```

Following is the query plan after creating index on `depart_station_id`:

```
 Limit  (cost=66.83..196.66 rows=10 width=28)
   ->  Result  (cost=66.83..560.17 rows=38 width=28)
         ->  Sort  (cost=66.83..66.93 rows=38 width=20)
               Sort Key: p1.depart_time
               ->  Hash Join  (cost=35.44..66.01 rows=38 width=20)
                     Hash Cond: (p2.trip_id = p1.trip_id)
                     Join Filter: (p1.depart_time < p2.arrive_time)
                     ->  Seq Scan on pass p2  (cost=0.00..30.06 rows=195 width=12)
                           Filter: (station_id = 1000)
                     ->  Hash  (cost=34.08..34.08 rows=109 width=12)
                           ->  Seq Scan on pass p1  (cost=0.00..34.08 rows=109 width=12)
                                 Filter: ((depart_time > '06:00:00'::time without time zone) AND
                                 ↪  (station_id = 1030))
         SubPlan 1
           ->  Aggregate  (cost=12.96..12.97 rows=1 width=8)
                 ->  Index Only Scan using idx_trip_id_on_reserved_ticket on reserved_ticket
                 ↪  (cost=0.43..12.58 rows=151 width=0)
                       Index Cond: ((trip_id = p1.trip_id) AND (arrive_station_id = 1000) AND
                       ↪  (depart_station_id = 1030) AND (travel_date >= '2023-08-01'::date)
                       ↪  AND (travel_date <= '2023-08-31'::date))

```

Following is the query plan after creating index on `arrive_station_id`:

```
Limit  (cost=66.83..196.66 rows=10 width=28)
  ->  Result  (cost=66.83..560.17 rows=38 width=28)
        ->  Sort  (cost=66.83..66.93 rows=38 width=20)
              Sort Key: p1.depart_time
              ->  Hash Join  (cost=35.44..66.01 rows=38 width=20)
                    Hash Cond: (p2.trip_id = p1.trip_id)
                    Join Filter: (p1.depart_time < p2.arrive_time)
                    ->  Seq Scan on pass p2  (cost=0.00..30.06 rows=195 width=12)
                          Filter: (station_id = 1000)
                    ->  Hash  (cost=34.08..34.08 rows=109 width=12)
                          ->  Seq Scan on pass p1  (cost=0.00..34.08 rows=109 width=12)
                                Filter: ((depart_time > '06:00:00'::time without time zone) AND
                                ↪  (station_id = 1030))
        SubPlan 1
          ->  Aggregate  (cost=12.96..12.97 rows=1 width=8)
                ->  Index Only Scan using idx_trip_id_on_reserved_ticket on reserved_ticket
                ↪  (cost=0.43..12.58 rows=151 width=0)
                      Index Cond: ((trip_id = p1.trip_id) AND (arrive_station_id = 1000) AND
                      ↪  (depart_station_id = 1030) AND (travel_date >= '2023-08-01'::date)
                      ↪  AND (travel_date <= '2023-08-31'::date))

```