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.

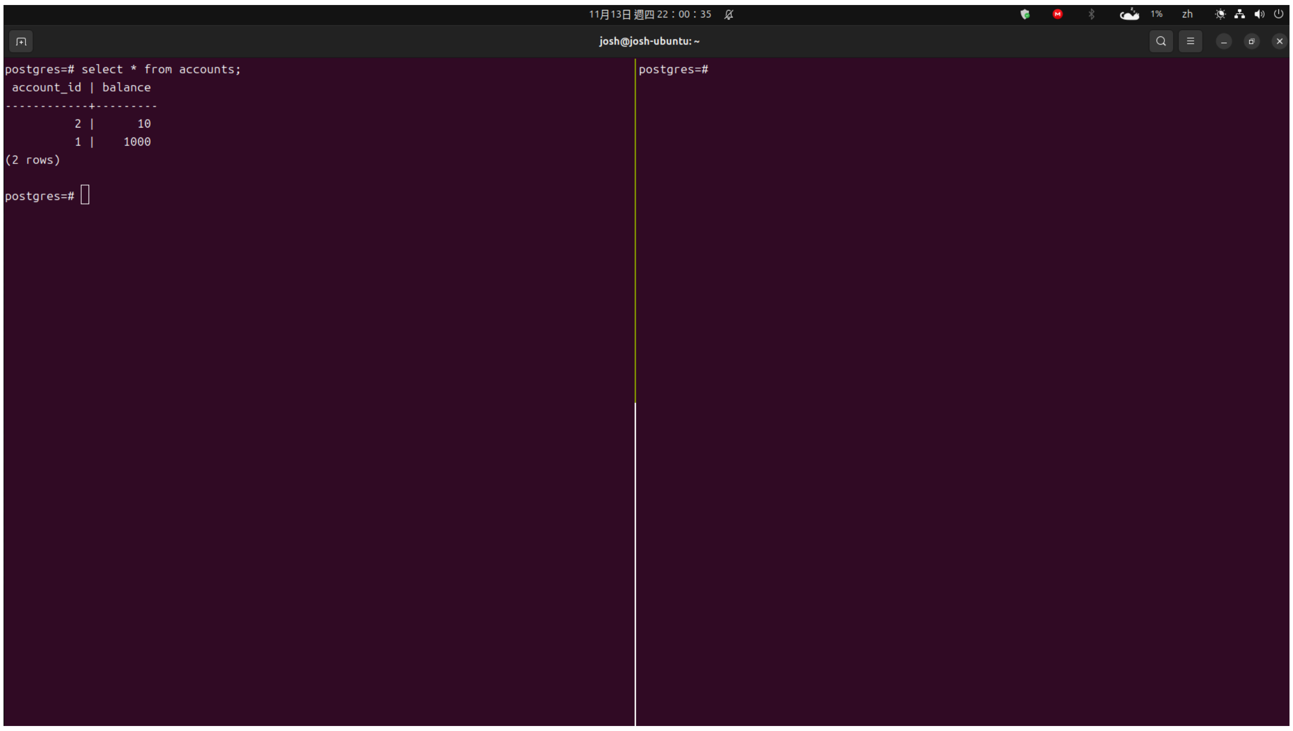
   (i) Transaction B reads the record of account_id 1.

   (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.

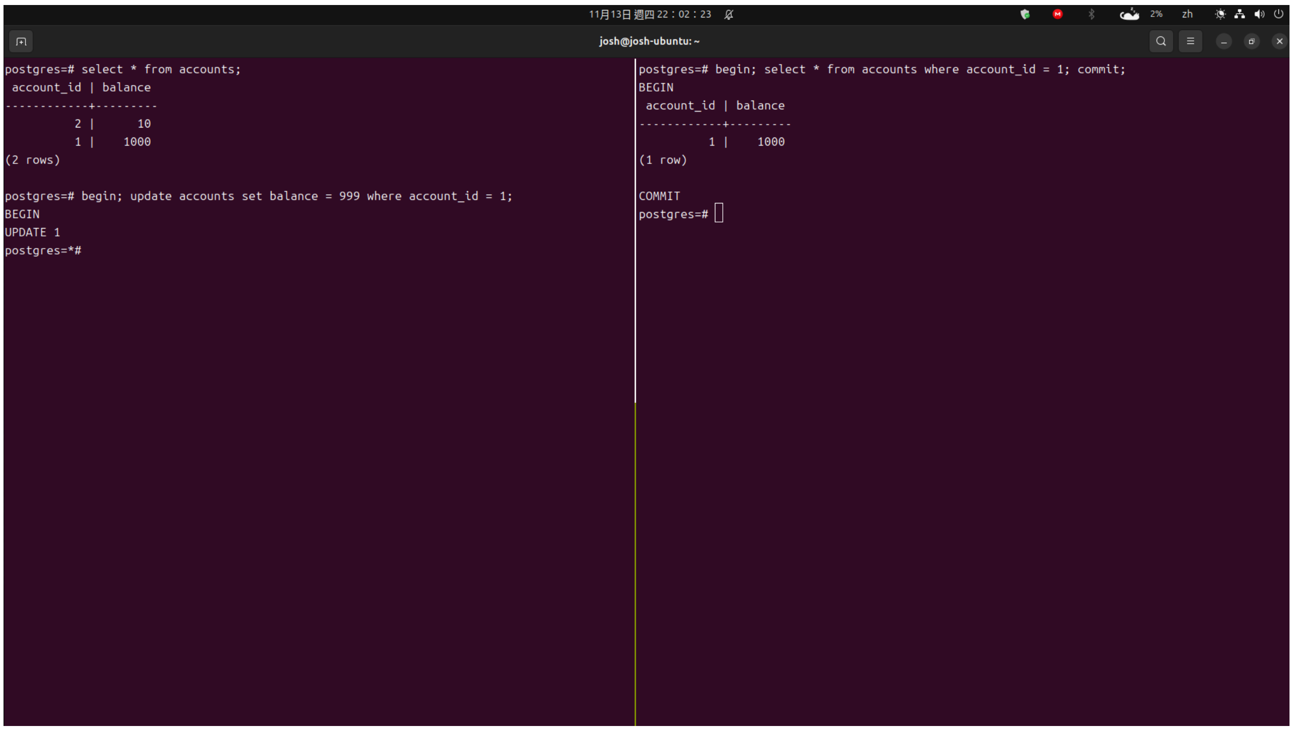


```
postgres=# select * from accounts;
 account_id | balance
------------+---------
          2 |      10
          1 |    1000
(2 rows)

postgres=#
```

```
postgres=#
```

Figure 1: Orginal status of the accounts table



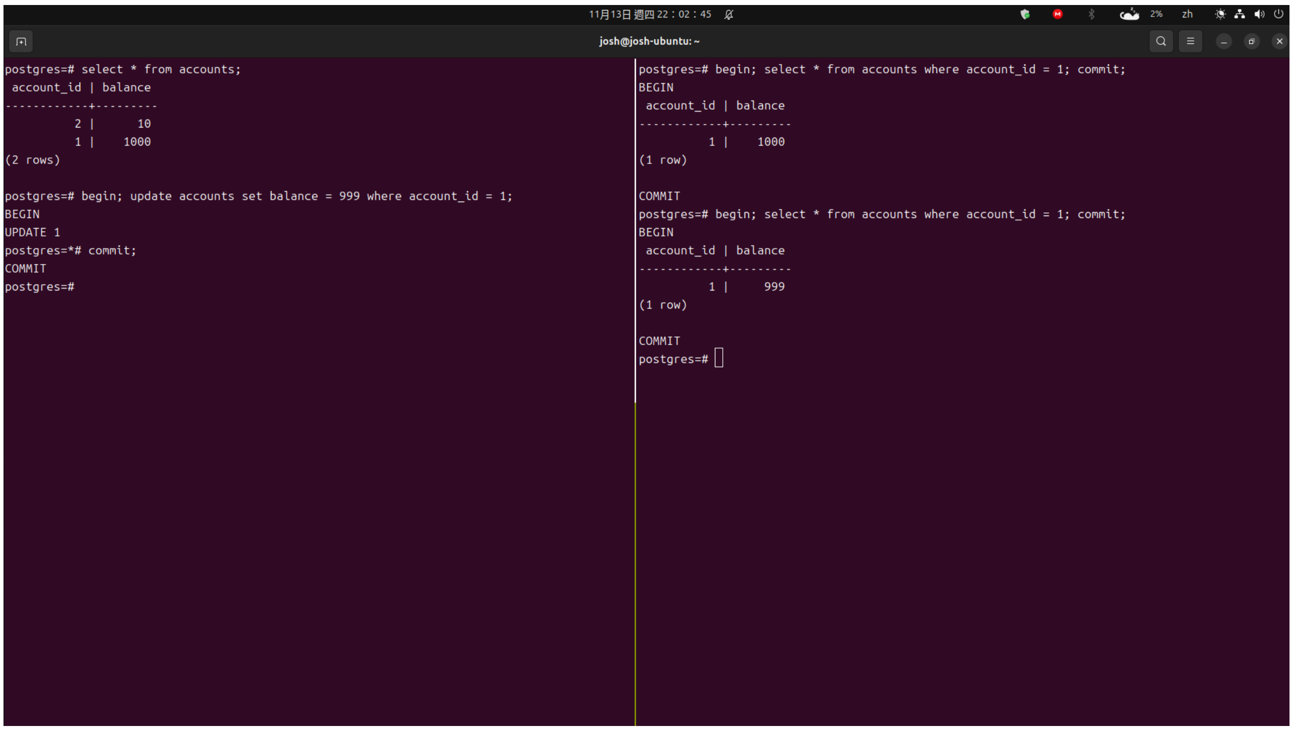
```
postgres=# select * from accounts;
 account_id | balance
------------+---------
          2 |      10
          1 |    1000
(2 rows)

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



```
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 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) (actual time=0.199..0.492 rows=10 loops=1)
  ->  Result  (cost=66.83..560.17 rows=38 width=28) (actual time=0.198..0.491 rows=10 loops=1)
        ->  Sort  (cost=66.83..66.93 rows=38 width=20) (actual time=0.145..0.146 rows=10
        ↪  loops=1)
              Sort Key: p1.depart_time
              Sort Method: top-N heapsort  Memory: 25kB
              ->  Hash Join  (cost=35.44..66.01 rows=38 width=20) (actual time=0.086..0.138
              ↪  rows=62 loops=1)
                    Hash Cond: (p2.trip_id = p1.trip_id)
                    Join Filter: (p1.depart_time < p2.arrive_time)
                    Rows Removed by Join Filter: 60
                    ->  Seq Scan on pass p2  (cost=0.00..30.06 rows=195 width=12) (actual
                    ↪  time=0.003..0.048 rows=195 loops=1)
                          Filter: (station_id = 1000)
                          Rows Removed by Filter: 1410
                    ->  Hash  (cost=34.08..34.08 rows=109 width=12) (actual time=0.071..0.072
                    ↪  rows=122 loops=1)
                          Buckets: 1024  Batches: 1  Memory Usage: 14kB
                          ->  Seq Scan on pass p1  (cost=0.00..34.08 rows=109 width=12) (actual
                          ↪  time=0.010..0.062 rows=122 loops=1)
                                Filter: ((depart_time > '06:00:00'::time without time zone) AND
                                ↪  (station_id = 1030))
                                Rows Removed by Filter: 1483
        SubPlan 1
          ->  Aggregate  (cost=12.96..12.97 rows=1 width=8) (actual time=0.034..0.034 rows=1
          ↪  loops=10)
                ->  Index Only Scan using idx_trip_id_on_reserved_ticket on reserved_ticket
                ↪  (cost=0.43..12.58 rows=151 width=0) (actual time=0.004..0.023 rows=507
                ↪  loops=10)
                      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))
                      Heap Fetches: 0
Planning Time: 0.191 ms
Execution Time: 0.510 ms
```

(c) Following is the code to create index on `trip_id`, `depart_station_id`, and `arrive_station_id`, `trip_id` apprears in both `pass` and `reserved_ticket` table, but the botttleneck is in the subquery which accesses `reserved_ticket` table, so I decide to create index on `trip_id` in `reserved_ticket` table.

```
create index idx_trip_id on reserved_ticket(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) (actual time=0.208..0.502 rows=10 loops=1)
  ->  Result  (cost=66.83..560.17 rows=38 width=28) (actual time=0.208..0.501 rows=10 loops=1)
        ->  Sort  (cost=66.83..66.93 rows=38 width=20) (actual time=0.153..0.154 rows=10
        ↪  loops=1)
              Sort Key: p1.depart_time
              Sort Method: top-N heapsort  Memory: 25kB
              ->  Hash Join  (cost=35.44..66.01 rows=38 width=20) (actual time=0.088..0.143
              ↪  rows=62 loops=1)
                    Hash Cond: (p2.trip_id = p1.trip_id)
                    Join Filter: (p1.depart_time < p2.arrive_time)
                    Rows Removed by Join Filter: 60
                    ->  Seq Scan on pass p2  (cost=0.00..30.06 rows=195 width=12) (actual
                    ↪  time=0.003..0.050 rows=195 loops=1)
                          Filter: (station_id = 1000)
                          Rows Removed by Filter: 1410
                    ->  Hash  (cost=34.08..34.08 rows=109 width=12) (actual time=0.070..0.071
                    ↪  rows=122 loops=1)
                          Buckets: 1024  Batches: 1  Memory Usage: 14kB
                          ->  Seq Scan on pass p1  (cost=0.00..34.08 rows=109 width=12) (actual
                          ↪  time=0.010..0.061 rows=122 loops=1)
                                Filter: ((depart_time > '06:00:00'::time without time zone) AND
                                ↪  (station_id = 1030))
                                Rows Removed by Filter: 1483
        SubPlan 1
          ->  Aggregate  (cost=12.96..12.97 rows=1 width=8) (actual time=0.034..0.034 rows=1
          ↪  loops=10)
                ->  Index Only Scan using idx_trip_id_on_reserved_ticket on reserved_ticket
                ↪  (cost=0.43..12.58 rows=151 width=0) (actual time=0.003..0.023 rows=507
                ↪  loops=10)
```

```
                      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))
                      Heap Fetches: 0
Planning Time: 0.955 ms
Execution Time: 0.521 ms
```

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

```
Limit  (cost=66.83..196.66 rows=10 width=28) (actual time=0.208..0.508 rows=10 loops=1)
  ->  Result  (cost=66.83..560.17 rows=38 width=28) (actual time=0.200..0.500 rows=10 loops=1)
        ->  Sort  (cost=66.83..66.93 rows=38 width=20) (actual time=0.149..0.150 rows=10 loops=1)
              Sort Key: p1.depart_time
              Sort Method: top-N heapsort  Memory: 25kB
              ->  Hash Join  (cost=35.44..66.01 rows=38 width=20) (actual time=0.086..0.139 rows=62 loops=1)
                    Hash Cond: (p2.trip_id = p1.trip_id)
                    Join Filter: (p1.depart_time < p2.arrive_time)
                    Rows Removed by Join Filter: 60
                    ->  Seq Scan on pass p2  (cost=0.00..30.06 rows=195 width=12) (actual time=0.003..0.049 rows=195 loops=1)
                          Filter: (station_id = 1000)
                          Rows Removed by Filter: 1410
                    ->  Hash  (cost=34.08..34.08 rows=109 width=12) (actual time=0.072..0.073 rows=122 loops=1)
                          Buckets: 1024  Batches: 1  Memory Usage: 14kB
                          ->  Seq Scan on pass p1  (cost=0.00..34.08 rows=109 width=12) (actual time=0.010..0.063 rows=122 loops=1)
                                Filter: ((depart_time > '06:00:00'::time without time zone) AND (station_id = 1030))
                                Rows Removed by Filter: 1483
        SubPlan 1
          ->  Aggregate  (cost=12.96..12.97 rows=1 width=8) (actual time=0.035..0.035 rows=1 loops=10)
                ->  Index Only Scan using idx_trip_id_on_reserved_ticket on reserved_ticket (cost=0.43..12.58 rows=151 width=0) (actual time=0.003..0.023 rows=507 loops=10)
                      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))
                      Heap Fetches: 0
Planning Time: 0.216 ms
Execution Time: 0.526 ms
```

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

```
Limit  (cost=66.83..196.66 rows=10 width=28) (actual time=0.207..0.512 rows=10 loops=1)
  ->  Result  (cost=66.83..560.17 rows=38 width=28) (actual time=0.206..0.511 rows=10 loops=1)
        ->  Sort  (cost=66.83..66.93 rows=38 width=20) (actual time=0.152..0.153 rows=10 loops=1)
              Sort Key: p1.depart_time
              Sort Method: top-N heapsort  Memory: 25kB
              ->  Hash Join  (cost=35.44..66.01 rows=38 width=20) (actual time=0.089..0.144 rows=62 loops=1)
                    Hash Cond: (p2.trip_id = p1.trip_id)
                    Join Filter: (p1.depart_time < p2.arrive_time)
                    Rows Removed by Join Filter: 60
                    ->  Seq Scan on pass p2  (cost=0.00..30.06 rows=195 width=12) (actual
                    ↪   time=0.003..0.050 rows=195 loops=1)
                          Filter: (station_id = 1000)
                          Rows Removed by Filter: 1410
                    ->  Hash  (cost=34.08..34.08 rows=109 width=12) (actual time=0.074..0.075
                    ↪   rows=122 loops=1)
                          Buckets: 1024  Batches: 1  Memory Usage: 14kB
                          ->  Seq Scan on pass p1  (cost=0.00..34.08 rows=109 width=12) (actual
                          ↪   time=0.010..0.065 rows=122 loops=1)
                                Filter: ((depart_time > '06:00:00'::time without time zone) AND
                                ↪   (station_id = 1030))
                                Rows Removed by Filter: 1483
        SubPlan 1
          ->  Aggregate  (cost=12.96..12.97 rows=1 width=8) (actual time=0.035..0.035 rows=1
          ↪   loops=10)
                ->  Index Only Scan using idx_trip_id_on_reserved_ticket on reserved_ticket
                ↪   (cost=0.43..12.58 rows=151 width=0) (actual time=0.004..0.023 rows=507
                ↪   loops=10)
                      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))
                      Heap Fetches: 0
Planning Time: 0.228 ms
Execution Time: 0.532 ms
```

We can observe that the estimated total cost remains the same (560.17) even after creating the single-column indexes. This is because the PostgreSQL query planner determined that a sequential scan was still cheaper than using any of the available indexes. Given the table size, data distribution, and the low selectivity of the filtering conditions, the planner estimated that scanning the entire table sequentially would cost less than performing an index scan. In this situation, the additional overhead of index traversal and random I/O did not yield any performance improvement, so the planner correctly chose a sequential scan.

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

```
create index idx_tda on reserved_ticket(trip_id, depart_station_id, arrive_station_id);
```

Following is the query plan after creating a composite index on `trip_id`, `depart_station_id`, and `arrive_station_id`:

```
Limit  (cost=66.83..196.66 rows=10 width=28) (actual time=0.217..0.512 rows=10 loops=1)
  ->  Result  (cost=66.83..560.17 rows=38 width=28) (actual time=0.212..0.507 rows=10 loops=1)
        ->  Sort  (cost=66.83..66.93 rows=38 width=20) (actual time=0.153..0.154 rows=10
        ↪   loops=1)
              Sort Key: p1.depart_time
              Sort Method: top-N heapsort  Memory: 25kB
              ->  Hash Join  (cost=35.44..66.01 rows=38 width=20) (actual time=0.089..0.144
              ↪   rows=62 loops=1)
                    Hash Cond: (p2.trip_id = p1.trip_id)
                    Join Filter: (p1.depart_time < p2.arrive_time)
```

```
          Rows Removed by Join Filter: 60
          ->  Seq Scan on pass p2  (cost=0.00..30.06 rows=195 width=12) (actual
          ↪  time=0.003..0.050 rows=195 loops=1)
             Filter: (station_id = 1000)
             Rows Removed by Filter: 1410
          ->  Hash  (cost=34.08..34.08 rows=109 width=12) (actual time=0.075..0.075
          ↪  rows=122 loops=1)
             Buckets: 1024  Batches: 1  Memory Usage: 14kB
             ->  Seq Scan on pass p1  (cost=0.00..34.08 rows=109 width=12) (actual
             ↪  time=0.010..0.065 rows=122 loops=1)
                Filter: ((depart_time > '06:00:00'::time without time zone) AND
                ↪  (station_id = 1030))
                Rows Removed by Filter: 1483
    SubPlan 1
     ->  Aggregate  (cost=12.96..12.97 rows=1 width=8) (actual time=0.035..0.035 rows=1
     ↪  loops=10)
        ->  Index Only Scan using idx_trip_id_on_reserved_ticket on reserved_ticket
        ↪  (cost=0.43..12.58 rows=151 width=0) (actual time=0.003..0.023 rows=507
        ↪  loops=10)
           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))
           Heap Fetches: 0
Planning Time: 0.246 ms
Execution Time: 0.531 ms
```

Still, it does not enhance the performance of query(estimated cost remains 560.17). The reason is similar to the previous analysis: despite the presence of a composite index, the PostgreSQL query planner determined that a sequential scan was still the most efficient method for accessing the data. The filtering conditions and data distribution likely resulted in low selectivity, making the overhead of using the index outweigh its benefits. Consequently, the planner opted for a sequential scan, leading to no change in the estimated cost.

5. (a) Fig. 4 shows the B+ tree index structure for the given data entries.

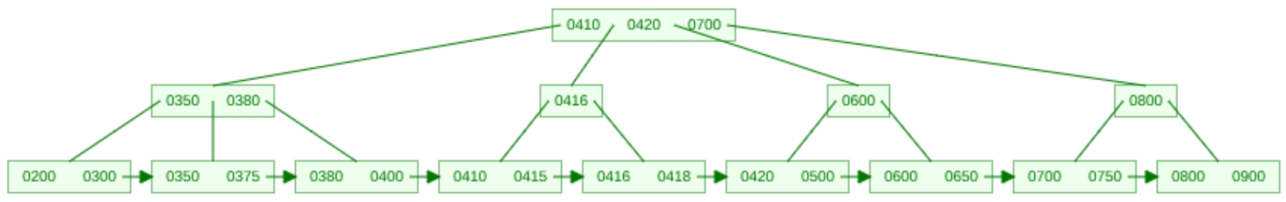


Figure 4: B+ Tree Index Structure

(b) Fig. 5 shows the before and after inserting 700 into the B+ tree. We can see that a new inner node is created after the insertion because the leaf node would be full after inserting 700. Thus it splits into two leaf nodes, and an inner node 700 is created to point to these two leaf nodes.

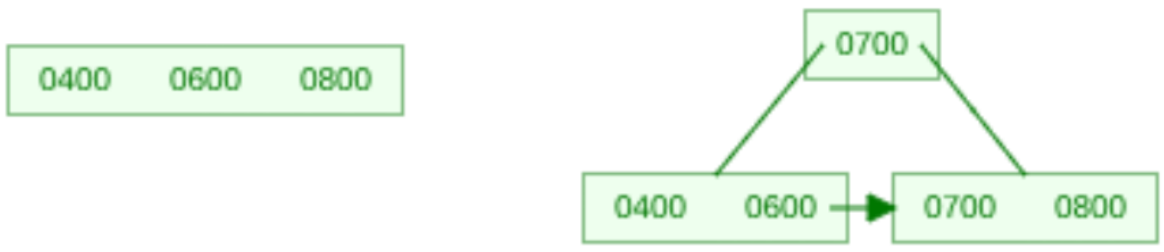


Figure 5: Before and after insert 700

(c) Fig. 6 shows the before and after inserting 350 into the B+ tree. As 350 is guided to the leftmost leaf node, and this leaf node is not full, we can simply insert 350 into this leaf node without any split.

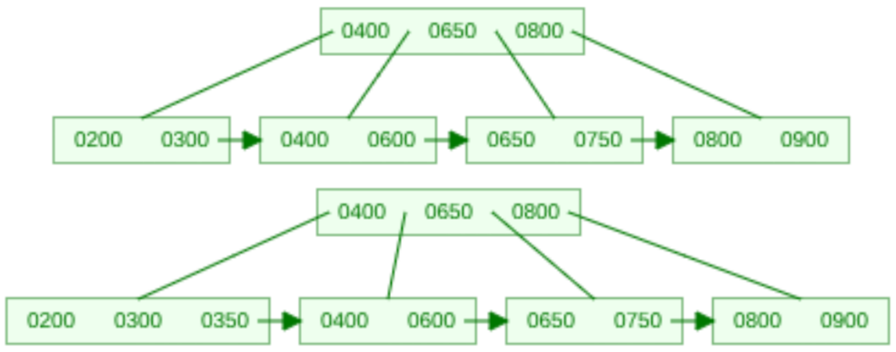


Figure 6: Before and after insert 350

(d) Fig. 7 shows the before and after inserting 418 into the B+ tree. As the leaf node where 418 is guided to would be full after inserting 418, it splits into two leaf nodes. And a new inner node 418 is created to point to these two leaf nodes.

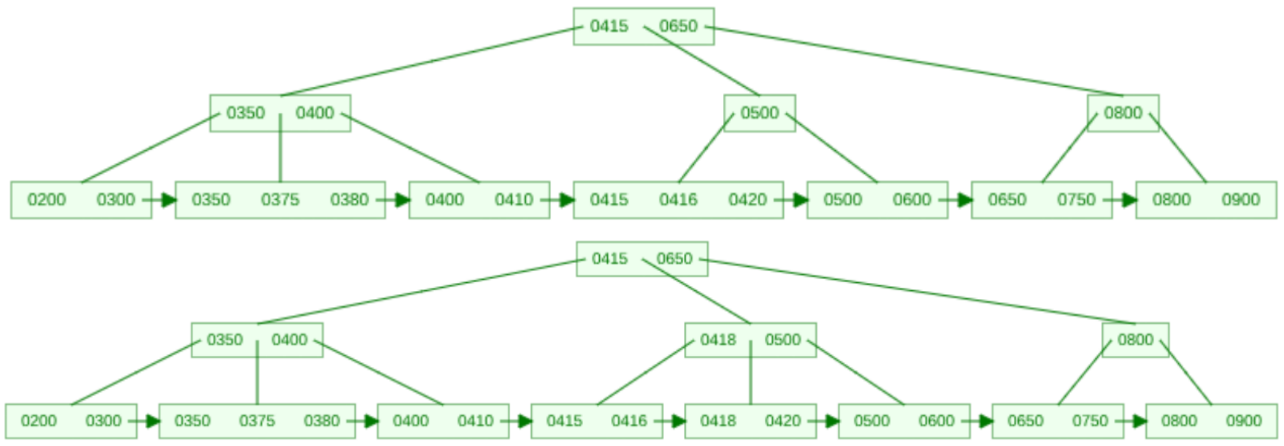


Figure 7: Before and after insert 418

6.
```sql
-- create index on pass(station_id, depart_time, trip_id)
create index index_sdt on pass(station_id, depart_time, trip_id);
```

```

-- explain analyze
 Limit  (cost=0.56..158.60 rows=10 width=28) (actual time=0.060..0.369 rows=10 loops=1)
   ->  Nested Loop  (cost=0.56..601.11 rows=38 width=28) (actual time=0.060..0.368 rows=10 loops=1)
         ->  Index Only Scan using index_sdt on pass p1  (cost=0.28..6.46 rows=109 width=12) (actual
         ↪  time=0.007..0.008 rows=18 loops=1)
               Index Cond: ((station_id = 1030) AND (depart_time > '06:00:00'::time without time
               ↪  zone))
               Heap Fetches: 0
         ->  Index Scan using pass_pkey on pass p2  (cost=0.28..0.92 rows=1 width=12) (actual
         ↪  time=0.001..0.001 rows=1 loops=18)
               Index Cond: ((trip_id = p1.trip_id) AND (station_id = 1000))
               Filter: (p1.depart_time < arrive_time)
               Rows Removed by Filter: 0
         SubPlan 1
           ->  Aggregate  (cost=12.96..12.97 rows=1 width=8) (actual time=0.034..0.034 rows=1
           ↪  loops=10)
                 ->  Index Only Scan using idx_trip_id_on_reserved_ticket on reserved_ticket
                 ↪  (cost=0.43..12.58 rows=151 width=0) (actual time=0.003..0.022 rows=507 loops=10)
                       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))
                       Heap Fetches: 0
 Planning Time: 0.220 ms
 Execution Time: 0.385 ms
```