資料庫管理 HW04

B12508026 戴偉璿

November 19, 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;
   2             select * from accounts where account_id = 1;
   3             commit;
   ```

   The execution steps are as follows:

   (a) Transaction A begins.

   (b) Transaction A updates balance to 1000 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 1000), 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 (1000) 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

1000 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 (1000) of account_id 1.

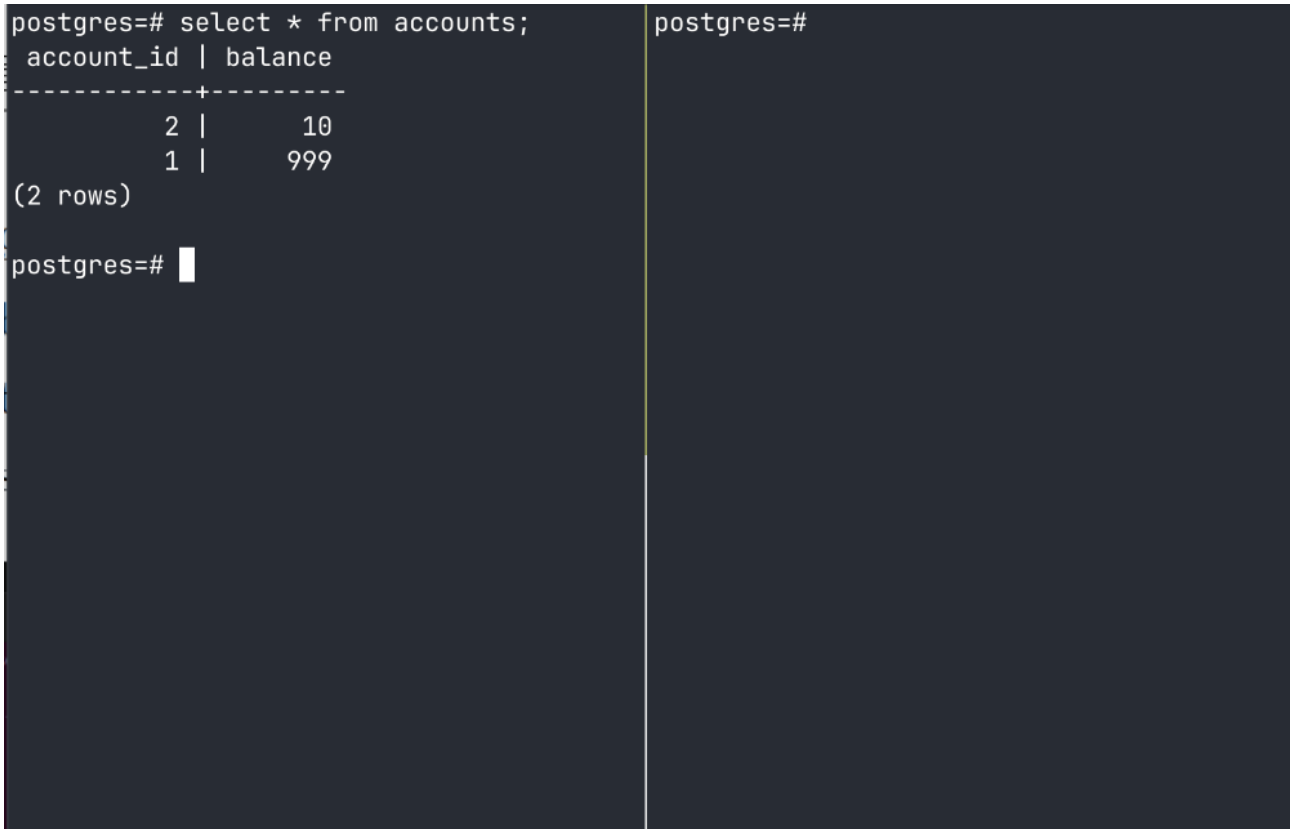

Figure 1: Orginal status of the accounts table

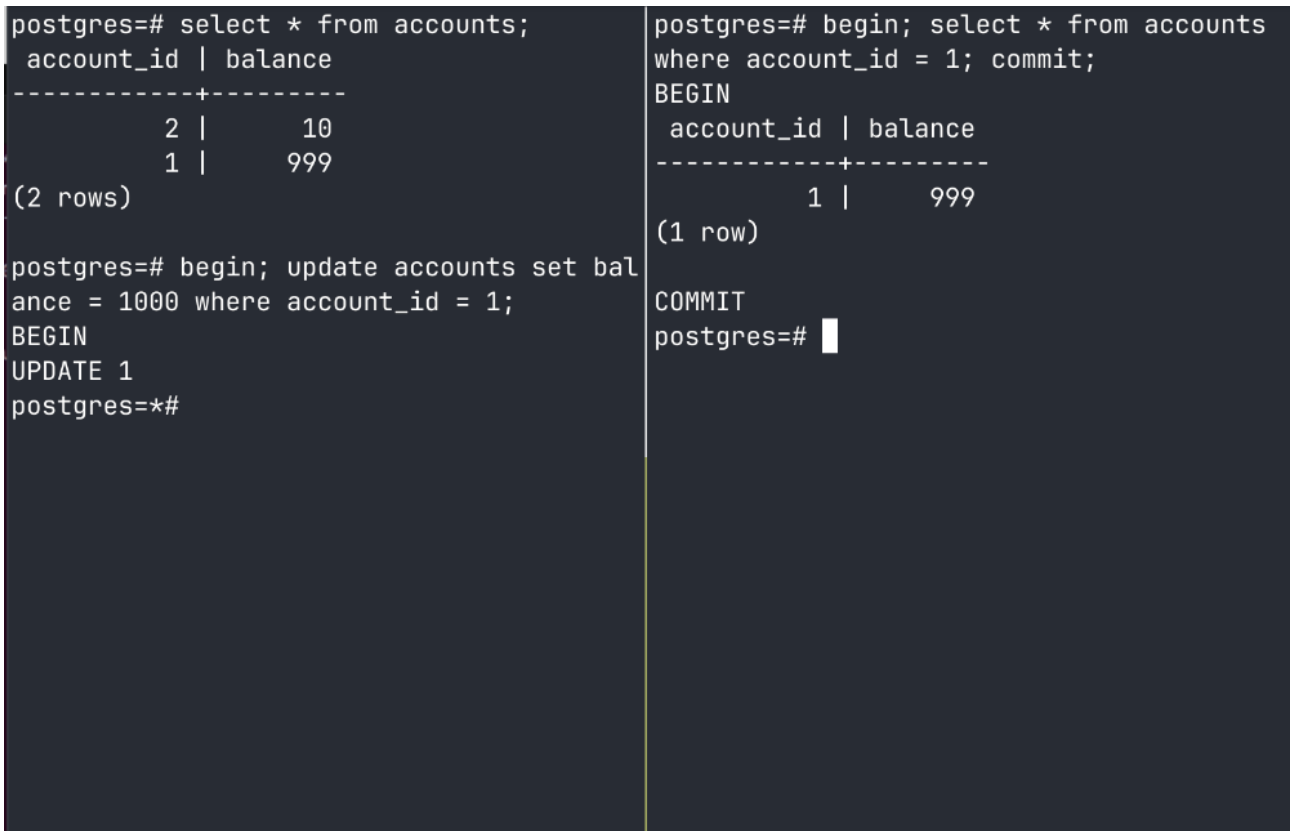

Figure 2: Transaction A updates balance to 1000 of account_id 1, but does not commit yet

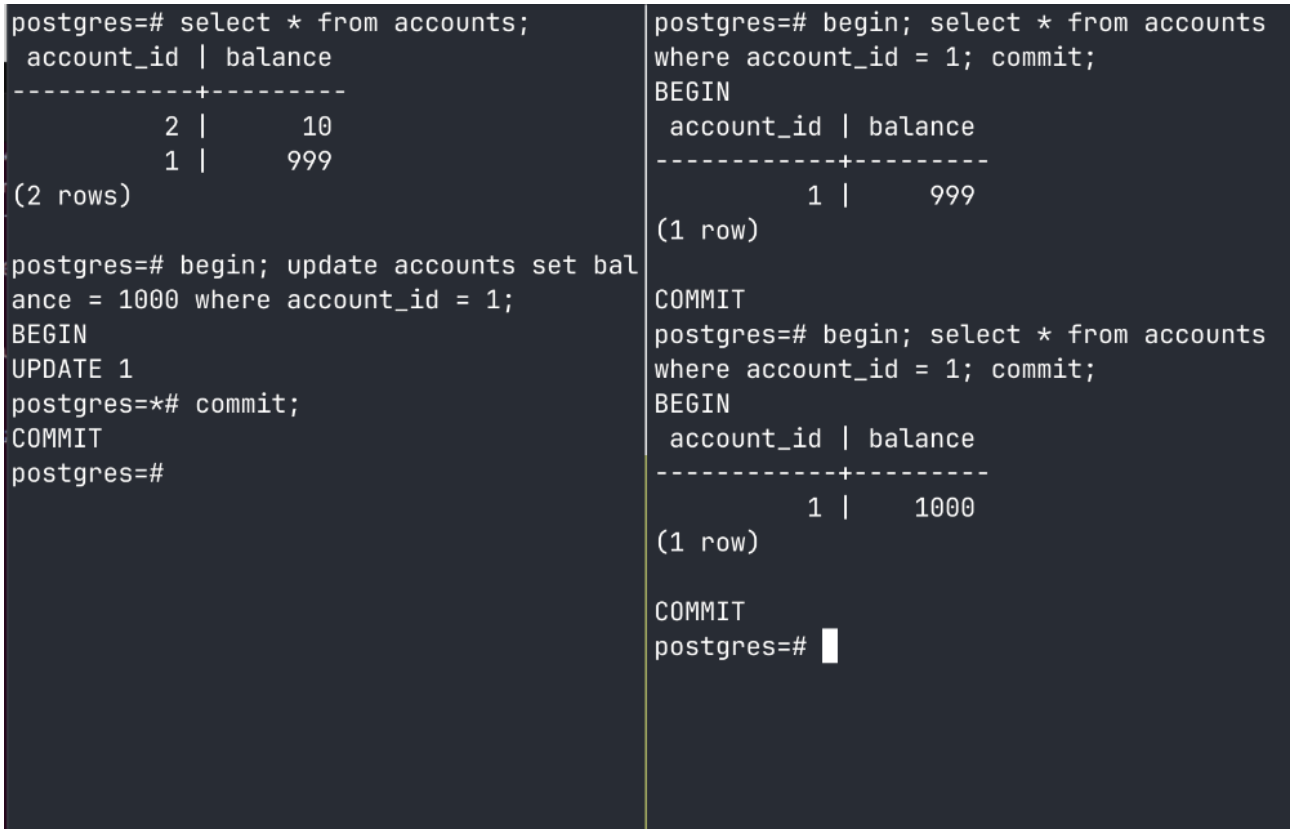

Figure 3: Transaction A commits, Transaction B reads the new balance of account_id 1

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.

   (d) Following is the 2PL table for the two transactions:

Table 1: 2PL tables for transactions T1 and T2

(a) Transaction T1 under 2PL

id	Operation	Phase
L_{11}	`read_lock(X)`	Expanding
O_{11}	`read_item(X)`	-
L_{12}	`read_lock(Y)`	Expanding
O_{12}	`read_item(Y)`	-
L_{13}	`read_lock(Z)`	Expanding
O_{13}	`read_item(Z)`	-
O_{14}	$Z \leftarrow X + Y$	-
L_{14}	`write_lock(Z)`	Expanding
O_{15}	`write_item(Z)`	-
O_{16}	$X \leftarrow 100$	-
L_{15}	`write_lock(X)`	Expanding
O_{17}	`write_item(X)`	-
L_{16}	`unlock(X)`	Shrinking
L_{17}	`unlock(Y)`	Shrinking
L_{18}	`unlock(Z)`	Shrinking

(b) Transaction T2 under 2PL

id	Operation	Phase
L_{21}	`read_lock(X)`	Expanding
O_{21}	`read_item(X)`	-
O_{22}	$X \leftarrow X + 10$	-
L_{22}	`write_lock(X)`	Expanding
O_{23}	`write_item(X)`	-
L_{23}	`read_lock(Z)`	Expanding
O_{24}	`read_item(Z)`	-
O_{25}	$Z \leftarrow Z + X$	-
O_{26}	`read_item(Z)`	-
L_{24}	`unlock(X)`	Shrinking
L_{25}	`unlock(Z)`	Shrinking

(e) Following is one possible conflict-serializable schedule that would lead to deadlock under 2PL:

Table 2: A non-serializable interleaving that leads to deadlock under 2PL

id	T1	T2	Phase
L11	`read_lock(X)`		Expanding
O11	`read_item(X)`		-
L21		`read_lock(X)`	Expanding
O21		`read_item(X)`	-
L22		`write_lock(X)`	**Blocked** (T1 holds X-R)
L12	`read_lock(Y)`		Expanding
O12	`read_item(Y)`		-
L13	`read_lock(Z)`		Expanding
O13	`read_item(Z)`		-
L14	`write_lock(Z)`		**Blocked** (T2 holds Z-R)

Deadlock: T_1 waits for Z while T_2 waits for X.
Wait-for cycle: $T_1 \rightarrow T_2 \rightarrow T_1$

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) Fig. 4 is hte query plan generated by PostgreSQL for the SQL statement above, the estimated cost is 3750384.97.

```
 Limit  (cost=66.83..986992.66 rows=10 width=28)
   →  Result  (cost=66.83..3750384.97 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=98692.56..98692.57 rows=1 width=8)
               →  Seq Scan on reserved_ticket  (cost=0.00..98692.18 rows=151 width=0)
                    Filter: ((travel_date ⩾ '2023-08-01'::date) AND (travel_date ⩽ '2023-08-31'::date) AND (trip_id = p1.trip_id) AND (depart_station
_id = 1030) AND (arrive_station_id = 1000))
 JIT:
   Functions: 27
   Options: Inlining true, Optimization true, Expressions true, Deforming true
(19 rows)
```

Figure 4: Query Plan without Index

   (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.

```sql
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);
```

Fig. 5 is the query plan after creating index on `trip_id`, Fig. 6 is the query plan after creating index on `depart_station_id`, Fig. 7 is the query plan after creating index on `arrive_station_id`.

For each index created, the estimated costs are as follows:

Index Created	Estimated Cost
no index	3750384.97
trip_id	1024884.52
depart_station_id	1559993.73
arrive_station_id	1731233.13

We can observe that creating an index on `trip_id` significantly reduces the estimated cost from 3750384.97 to 1024884.52, indicating a substantial improvement in query performance. However, creating indexes on `depart_station_id` and `arrive_station_id` also reduces the estimated costs, but not as significantly as the `trip_id` index. This suggests that while these indexes do help, they are not as effective in optimizing the query as the

```
Limit  (cost=66.83..269755.70 rows=10 width=28)
  →  Result  (cost=66.83..1024884.52 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=26968.86..26968.87 rows=1 width=8)
              →  Bitmap Heap Scan on reserved_ticket  (cost=163.24..26968.49 rows=151 width=0)
                   Recheck Cond: (trip_id = p1.trip_id)
                   Filter: ((travel_date ≥ '2023-08-01'::date) AND (travel_date ≤ '2023-08-31'::date) AND (depart_station_id = 1030) AND (arrive_sta
tion_id = 1000))
                   →  Bitmap Index Scan on idx_trip_id  (cost=0.00..163.20 rows=14770 width=0)
                        Index Cond: (trip_id = p1.trip_id)
JIT:
  Functions: 30
  Options: Inlining false, Optimization false, Expressions true, Deforming true
(22 rows)
```

Figure 5: Query Plan with trip_id Index

```
Limit  (cost=66.83..410573.91 rows=10 width=28)
  →  Result  (cost=66.83..1559993.73 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=41050.69..41050.70 rows=1 width=8)
              →  Bitmap Heap Scan on reserved_ticket  (cost=2553.93..41050.31 rows=151 width=0)
                   Recheck Cond: (depart_station_id = 1030)
                   Filter: ((travel_date ≥ '2023-08-01'::date) AND (travel_date ≤ '2023-08-31'::date) AND (trip_id = p1.trip_id) AND (arrive_station
_id = 1000))
                   →  Bitmap Index Scan on idx_depart_station_id  (cost=0.00..2553.89 rows=234328 width=0)
                        Index Cond: (depart_station_id = 1030)
JIT:
  Functions: 30
  Options: Inlining false, Optimization false, Expressions true, Deforming true
(22 rows)
```

Figure 6: Query Plan with depart_station_id Index

```
Limit  (cost=66.83..455636.91 rows=10 width=28)
  →  Result  (cost=66.83..1731233.13 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=45556.99..45557.00 rows=1 width=8)
              →  Bitmap Heap Scan on reserved_ticket  (cost=4022.50..45556.61 rows=151 width=0)
                   Recheck Cond: (arrive_station_id = 1000)
                   Filter: ((travel_date ≥ '2023-08-01'::date) AND (travel_date ≤ '2023-08-31'::date) AND (trip_id = p1.trip_id) AND (depart_station
_id = 1030))
                   →  Bitmap Index Scan on idx_arrive_station_id  (cost=0.00..4022.46 rows=369338 width=0)
                        Index Cond: (arrive_station_id = 1000)
JIT:
  Functions: 30
  Options: Inlining false, Optimization false, Expressions true, Deforming true
(22 rows)
```

Figure 7: Query Plan with arrive_station_id Index

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

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

Fig. 8 is the query plan after creating a composite index on `trip_id`, `depart_station_id`, and `arrive_station_id`:

```
Limit  (cost=66.83..5902.97 rows=10 width=28)
  →  Result  (cost=66.83..22244.14 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=583.59..583.60 rows=1 width=8)
              →  Bitmap Heap Scan on reserved_ticket  (cost=6.36..583.21 rows=151 width=0)
                   Recheck Cond: ((trip_id = p1.trip_id) AND (depart_station_id = 1030) AND (arrive_station_id = 1000))
                   Filter: ((travel_date ⩾ '2023-08-01'::date) AND (travel_date ⩽ '2023-08-31'::date))
                   →  Bitmap Index Scan on idx_tda  (cost=0.00..6.32 rows=151 width=0)
                        Index Cond: ((trip_id = p1.trip_id) AND (depart_station_id = 1030) AND (arrive_station_id = 1000))
(19 rows)
```

Figure 8: Query Plan with Composite Index

5. (a) Fig. 9 shows the B+ tree index structure for the given data entries.

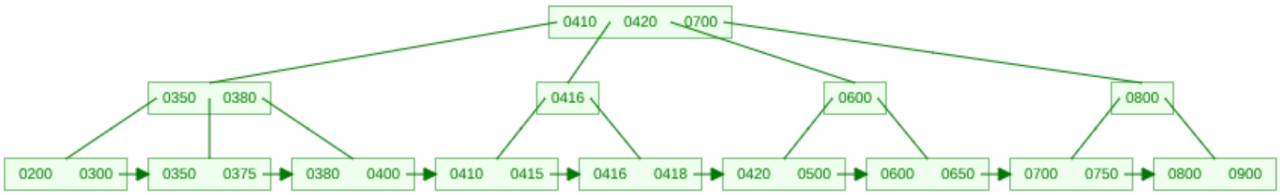

Figure 9: B+ Tree Index Structure

(b) Fig. 10 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 10: Before and after insert 700

(c) Fig. 11 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.

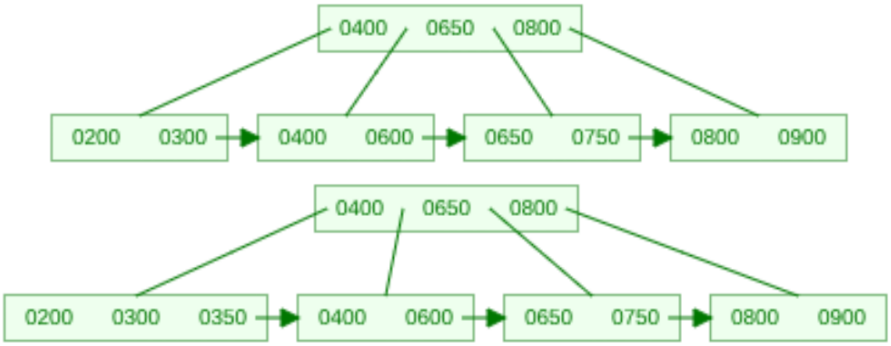

Figure 11: Before and after insert 350

(d) Fig. 12 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.

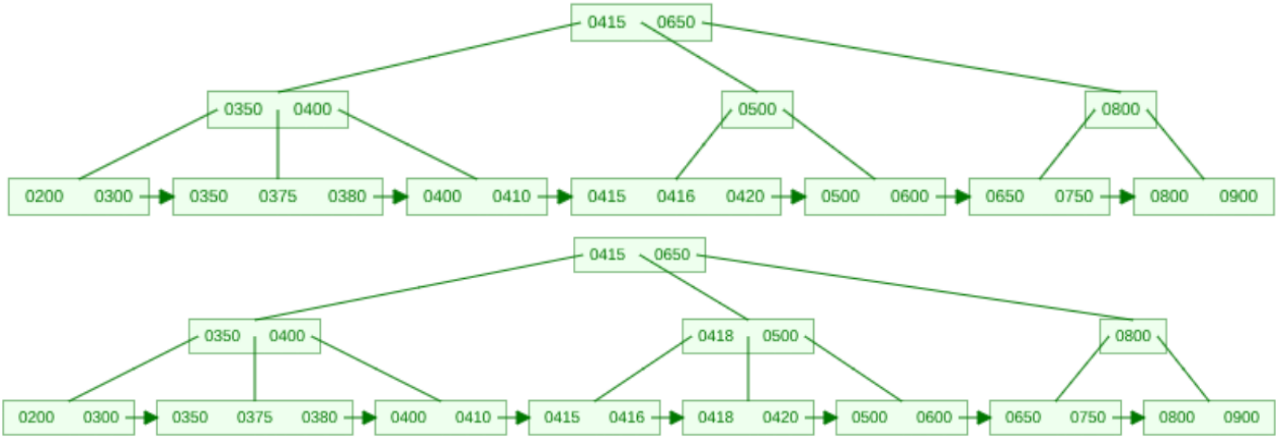

Figure 12: Before and after insert 418

(e) Fig. 13 shows the B+ tree generated by buttom-up method.

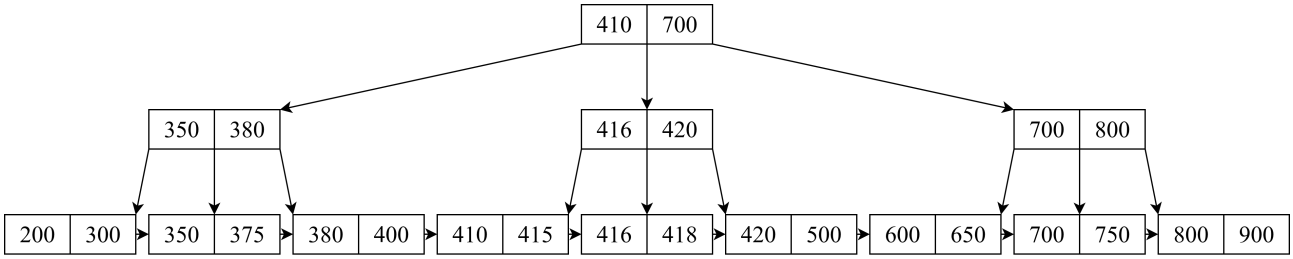

Figure 13: B+ Tree generated by buttom-up method

6. After observing that using a multicolumn index already led to a noticeably lower estimated cost compared to the single-column indexes, I further attempted to include addi-

tional filtering columns from the query in order to reduce the search space even more. In PostgreSQL's B-tree multicolumn indexes, the order of the indexed columns is crucial because the query optimizer can only take full advantage of the index when the filtering conditions match the *leftmost prefix* of the index. In this query, the predicates on `trip_id` and `arrive_station_id` exhibit higher selectivity than those on `depart_station_id` and `travel_date`. Therefore, placing the columns in the order `(trip_id, arrive_station_id, depart_station_id, travel_date)` allows PSQL to narrow down the index search range earlier, which leads to a significantly lower estimated cost compared to other column orderings, even if they contain the same set of columns.

Following is the code to create the optimized multicolumn index:

```
1 create index idx on reserved_ticket(trip_id, arrive_station_id, depart_station_id, travel_date);
```

The query plan after creating the index is shown in Fig. 14:

```
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_opt 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))
(16 rows)
```

Figure 14: Query Plan after creating optimized index

We can see that the estimated cost is further reduced to 560.17 after creating the optimized index, which is significantly lower than the previous costs. This indicates that the new index is highly effective in improving the query performance for the given SQL statement.