資料庫管理 HW03

B12508026 戴偉璿

1. (a) Left join all advisors(e) and their advisees(s), if someone has no advisee, then s would be NULL.

```sql
select e.id, e.name from employee as e
left join employee s on e.id=s.supervisor_id
where s.supervisor_id is null;
```

Fig 1 is the result:

id	name
E972346850	Hao-Tien Chu
A521870632	Ching-Yi Chao
A473564811	Chih-Yuan Lee
A671224980	Chieh Sun
S153004821	Po-Lin Chen

Figure 1: Result of 1(a)

(b) Find the latest store id of each employee before 2025-01-05 and left join to the employee table.

```sql
select e.id as employee_id, h.store_id
from employee e
left join employee_store_history h
on e.id = h.employee_id
and h.start_date_time=(
    select max(h2.start_date_time)
    from employee_store_history h2
    where h2.employee_id = e.id
        and h2.start_date_time <= '2025-01-05'
    );
```

Fig 2 is the result:

employee_id	store_id
L645977505	0
A465113807	1
S657432198	1
A473564811	1
A521870632	1
A671224980	2
S153004821	2
E972346850	2

Figure 2: Result of 1(b)

(c) Using `limit 1` to obtain the first store id and `limit 1 offset 1` to obtain the second store id (after skipping the first one), then join them to produce the final result.

```sql
select e.id as employee_id,
    (select h1.store_id from employee_store_history h1 where h1.employee_id=e.id order by
    ↪ h1.start_date_time limit 1) as first_store_id,
    (select h2.store_id from employee_store_history h2 where h2.employee_id=e.id order by
    ↪ h2.start_date_time limit 1 offset 1) as second_store_id
from employee e;
```

Fig 3 is the result:

employee_id	first_store_id	second_store_id
L645977505	0	
A465113807	1	
S657432198	2	1
A473564811	1	
A521870632	1	
A671224980	1	2
S153004821	2	
E972346850	2	

Figure 3: Result of 1(c)

(d) Calculating the total quantity purchased for each product, then ordering by total quantity and product_id, using `limit 2 offset 3` to find the 4th and 5th prod-

ucts. Finally, joining with purchase_detail and purchase tables to get the required information.

```sql
with total_qty as(
    select pd.product_id as product_id, sum(pd.qty) as total_qty, count(*) as purchase_count
    from purchase_detail pd
    group by pd.product_id
),
target_product as(
    select product_id
    from total_qty
    order by total_qty desc, product_id asc
    limit 2 offset 3
)
select p.id as product_id, p.name as product_name, pu.store_id as store_id, count(*) as
↪  purchase_count, sum(pd.qty) as total_qty
from target_product tp
join product p on tp.product_id = p.id
join purchase_detail pd on pd.product_id = p.id
join purchase pu on pd.purchase_no = pu.purchase_no
group by p.id, p.name, pu.store_id
order by p.id, pu.store_id;
```

Fig 4 is the result:

product_id	product_name	store_id	purchase_count	total_qty
2	Skin Moisturizing Lotion	1	1	300
2	Skin Moisturizing Lotion	2	1	250
4	Baby Comfort Diapers	1	2	350

Figure 4: Result of 1(d)

(e) Similar to 1(d), but using cross join to get all combinations of the target products and stores, then left join with purchase and purchase_detail to get the required information (if no purchase, then 0).

```sql
with total_qty as (
    select pd.product_id, sum(pd.qty) as total_qty, count(*) as purchase_count
    from purchase_detail pd
    group by pd.product_id
),
target_product as (
    select product_id
    from total_qty
    order by total_qty desc, product_id asc
    limit 2 offset 3
)
select
    p.id as product_id, p.name as product_name, s.id as store_id, coalesce(count(pd.product_id),
    ↪  0) as cnt, coalesce(sum(pd.qty), 0) as total_qty
from target_product tp
join product p on tp.product_id = p.id
cross join store s
left join purchase pu on pu.store_id = s.id
```

```
left join purchase_detail pd on pd.purchase_no = pu.purchase_no and pd.product_id = p.id
group by p.id, p.name, s.id
order by p.id, s.id;

```

Fig 5 is the result:

product_id	product_name	store_id	cnt	total_qty
2	Skin Moisturizing Lotion	0	0	0
2	Skin Moisturizing Lotion	1	1	300
2	Skin Moisturizing Lotion	2	1	250
4	Baby Comfort Diapers	0	0	0
4	Baby Comfort Diapers	1	2	350
4	Baby Comfort Diapers	2	0	0

Figure 5: Result of 1(e)

(f) I first calculate the total spending amount of each member in each store, then cross join all stores and members to get all combinations, and left join with the previous result to get the total spending amount (if no spending, then 0). Finally, I rank the members in each store by their total spending amount.

```
with member_spending as (
    select sa.store_id as store_id, sa.member_id as member_id, sum(sd.unit_price*sd.qty) as
    ↪  amount
    from sales sa
    join sales_detail sd on sd.receipt_no = sa.receipt_no
    where member_id is not null
    group by sa.store_id, sa.member_id
)
select s.id as store_id, m.id as member_id, coalesce(ms.amount, 0) as total_amount, rank()
↪  over(partition by s.id order by coalesce(ms.amount, 0) desc) as rnk
from store s
cross join member m
left join member_spending ms on ms.store_id = s.id and ms.member_id = m.id
order by s.id, rnk, m.id;
```

Fig 6 is the result:

store_id	member_id	total_amount	rnk
0	1	0	1
0	2	0	1
0	3	0	1
0	4	0	1
0	5	0	1
0	6	0	1
0	7	0	1
0	8	0	1
0	9	0	1
0	10	0	1
1	9	8605	1
1	8	1966	2
1	3	920	3
1	1	505	4
1	6	190	5
1	5	110	6
1	2	0	7
1	4	0	7
1	7	0	7
1	10	0	7
2	2	2695	1
2	7	1291	2
2	10	565	3
2	4	332	4
2	1	0	5
2	3	0	5
2	5	0	5
2	6	0	5
2	8	0	5
2	9	0	5

Figure 6: Result of 1(f)

2. (a) First, I identified the hard questions based on the number of total answers and correct answers. Then, for each hard question, I found each user's first correct submission and ranked the users by cost time and submission timestamp. Finally, I counted the number of hard questions where each user ranked in the top 3 and kept only those who appeared in at least two questions.

   Fig. 7 shows the leaderboard of top users appearing in at least two hard questions.

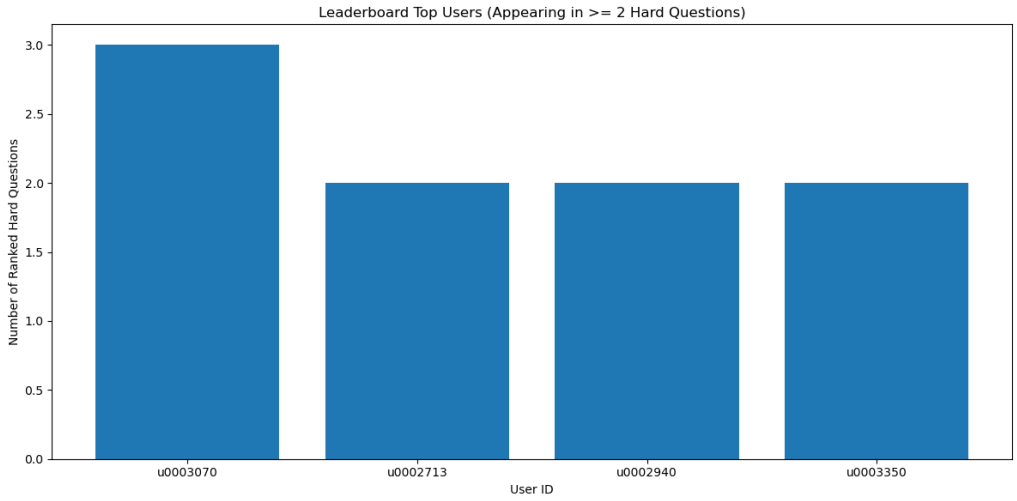

Figure 7: Leaderboard of Top Users Appearing in at Least Two Hard Questions

Following is my source code, I used DuckDB to execute the SQL query and Matplotlib to visualize the results.

```python
import duckdb
import matplotlib.pyplot as plt

con = duckdb.connect('my_database.db')

df = con.sql("""
-- list the hard questions
with hard_questions as(
    select question_id as id, count(*) as total_ans, sum(is_correct) as total_ac
    from answers
    group by question_id
    having total_ans >= 1000 and total_ac <= 500
),-- list the rank for all users
first_correct_pre as(
    select question_id, user_id, cost_time, created_at, rank() over(partition by question_id,
    ↪  user_id order by created_at asc) as rnk
    from answers
    where is_correct = 1
),-- first correct for each user
first_correct as(
    select question_id, user_id, cost_time, created_at
    from first_correct_pre
    where rnk = 1
),--rank users for hard questions
usr_rnk as(
    select question_id, user_id, rank() over(partition by question_id order by cost_time asc,
    ↪  created_at asc) as rnk
    from first_correct fc
    join hard_questions hq on hq.id = fc.question_id
)
select user_id, count(distinct question_id) as appr_cnt
from usr_rnk
where rnk <= 3
group by user_id
having appr_cnt >= 2
order by appr_cnt desc, user_id asc;
```

```
35 """).df()
36 print(df)
37
38 plt.figure(figsize=(12,6))
39 plt.bar(df["user_id"].astype(str), df["appr_cnt"])
40 plt.xlabel("User ID")
41 plt.ylabel("Number of Ranked Hard Questions")
42 plt.title("Leaderboard Top Users (Appearing in >= 2 Hard Questions)")
43 plt.tight_layout()
44 plt.savefig("2a_leaderboard.png")
45 plt.show()
```

(b) Using the same CTEs as in 2a to find the users who ranked in the top 3 for hard questions. Then I calculated their scores based on the given scoring system. Finally, I kept only those users with a total score of at least 5.
Fig. 8 shows the leaderboard of top users with a score of at least 5.

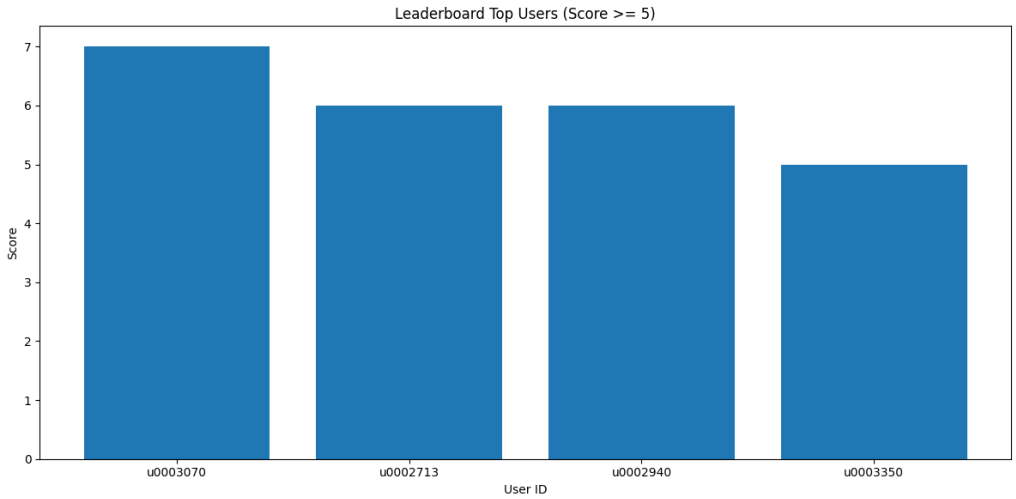

Figure 8: Leaderboard of Top Users with a Score of at Least 5

Following is my source code:

```
1  import duckdb
2  import matplotlib.pyplot as plt
3
4  con = duckdb.connect('my_database.db')
5
6  df = con.sql("""
7  -- list the hard questions
8  with hard_questions as(
9      select question_id as id, count(*) as total_ans, sum(is_correct) as total_ac
10     from answers
11     group by question_id
12     having total_ans >= 1000 and total_ac <= 500
13 ),-- list the rank for all users
14 first_correct_pre as(
15     select question_id, user_id, cost_time, created_at, rank() over(partition by question_id,
       ↪  user_id order by created_at asc) as rnk
16     from answers
```

```
    where is_correct = 1
),-- first correct for each user
first_correct as(
    select question_id, user_id, cost_time, created_at
    from first_correct_pre
    where rnk = 1
),--rank users for hard questions
usr_rnk as(
    select question_id, user_id, rank() over(partition by question_id order by cost_time asc,
    ↪  created_at asc) as rnk
    from first_correct fc
    join hard_questions hq on hq.id = fc.question_id
)
select user_id,
       sum(
           cast(rnk = 1 as int) * 4 +
           cast(rnk = 2 as int) * 2 +
           cast(rnk = 3 as int) * 1
       ) as score
from usr_rnk
where rnk <= 3
group by user_id
having score >= 5
order by score desc, user_id asc;
""").df()
print(df)

plt.figure(figsize=(12,6))
plt.bar(df["user_id"].astype(str), df["score"])
plt.xlabel("User ID")
plt.ylabel("Score")
plt.title("Leaderboard Top Users (Score >= 5)")
plt.tight_layout()
plt.savefig("2b_leaderboard.png")
plt.show()
```

3. (a) i. I'll use **Surrogate Key** as the primary key because Nature Key might be ambiguous. For example, multiple vehicles may be scheduled at the same time, and a composite Natural Key (such as vehicle ID, departure station, destination station, and time) would be too complex and hard to maintain.
   I'll use **Serial Number** instead of UUID because dispatch record only exists in a single table and does not require global uniqueness. Serial numbers are simpler, easier to manage, and provide better indexing performance for frequent insertions.

   ii. Yes, I' ll add a "soft delete flag" to the dispatch record table. Dispatch records are important historical data that may be needed for future analysis or auditing, so they should not be physically deleted.
   Soft deletion allows us to retain the record while marking it as inactive, preventing accidental data loss and preserving referential integrity with related tables. It also provides flexibility to restore records if needed.

iii. As Fig 9 shows:

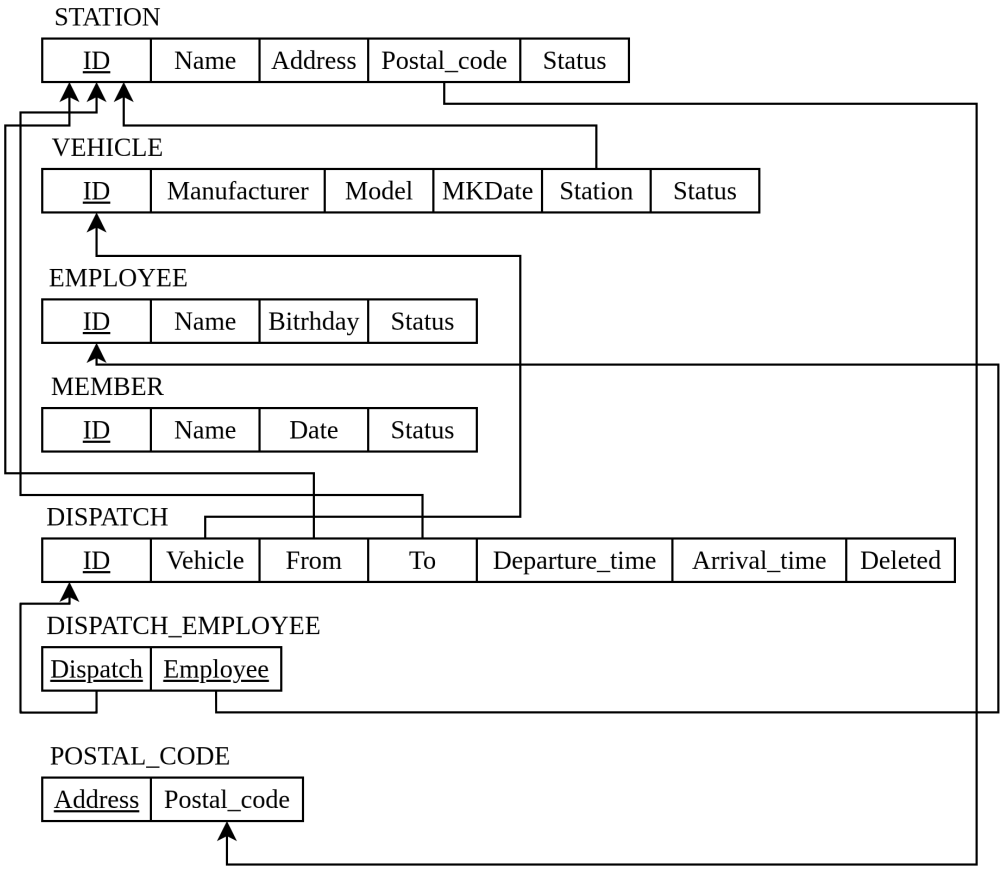

Figure 9: Relational Schema Diagram for Dispatch Record

(b) As Fig 10 shows:

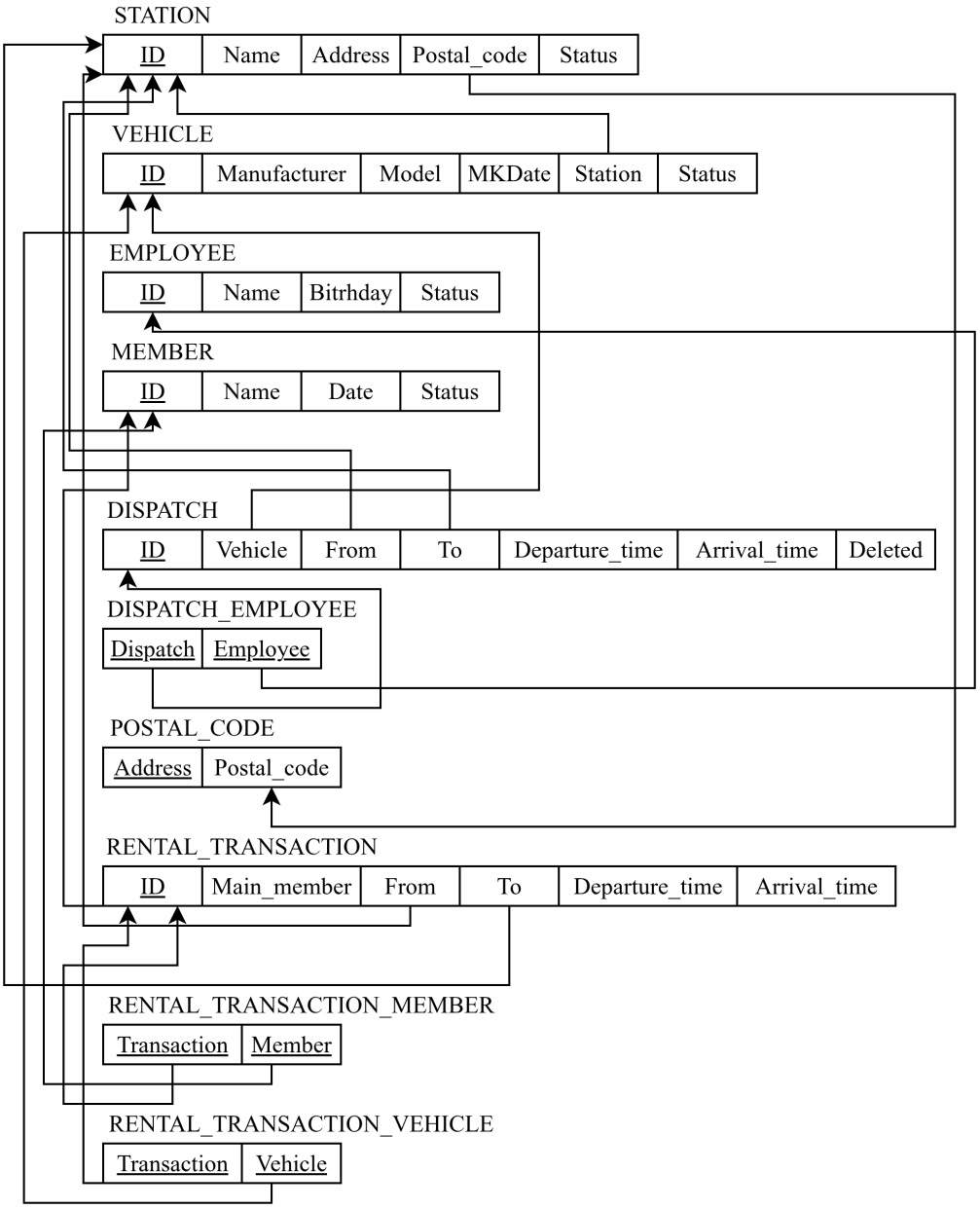

Figure 10: Relational Schema Diagram for Dispatch and Rental Record

4. (a) As Table 1 shows. I know that if it violates a lower normal form, it must violate all higher normal forms. But this table only discusses each normal form separately.

NF	Check	Reason
1	No	It has multiple employees recorded in one table
2	No	`make_date` only related to the `v_id`, but the primary key is composite with `v_id` and `from_datetime`.
3	No	We can get `v_id` by `(v_id, from_datetime)` and `make_date` from `v_id`.
BC	No	As it violates 3NF, it must also violate BCNF(because `v_id` is not a super key)
4	Yes	

Table 1: Normal Form Check for Vehicle Maintenance Record

(b) Table 2 shows the normal form checking of `RELOCATION`, and `RELOCATION_EMPLOYEE` does not violate any normal form. I know that if it violates a lower normal form, it must violate all higher normal forms. But this table only discusses each normal form separately.

NF	Check	Reason
1	Yes	
2	Yes	
3	No	In `RELOCATION` table, `make_date` is related to `v_id` and `v_id` is not a primary key.
BC	No	As it violates 3NF, it must also violate BCNF(because `v_id` is not a super key)
4	Yes	

Table 2: Normal Form Check for Vehicle Maintenance Record