Database Management Homework 3

Po-Yen Chu

1 Question 1

(a) According to the question, the SQL code is provided below, and the result is presented as Table 1.

```
WITH multi_dnum AS ( --sort out department with multiple
   locations
    SELECT dnumber
    FROM public.dept_locations
    GROUP BY dnumber
    HAVING COUNT(dnumber) > 1
), dependent_count AS ( --calculate dependent
    SELECT essn, COUNT(essn) AS dependent_count
    FROM public.dependent
    GROUP BY essn
)

SELECT ssn, bdate AS birth_date, COALESCE(dependent_count.
   dependent_count, 0) AS dependent_count
FROM public.employee
LEFT JOIN dependent_count ON ssn = essn
WHERE dno IN (SELECT dnumber FROM multi_dnum)
```

index	ssn	birth_date	dependent_count
1	123456789	1965-01-09	3
2	333445555	1955-12-08	3
3	453453453	1972-07-31	0
4	666884444	1962-09-15	0

Table 1: Information of Employee's Department Locating in Multiple Locations

(b) According to the question, the SQL code is provided below, and the result is presented as Table 2.

```sql
WITH employee_info AS (
    SELECT dno,
           AVG(EXTRACT(YEAR FROM AGE('2024-10-1', bdate)))
               AS avg_age
    FROM public.employee
    GROUP BY dno
    HAVING COUNT(dno) > 1
)

SELECT dnumber AS department_id, dname AS department_name,
    avg_age
FROM public.department
LEFT JOIN employee_info ON dnumber=dno
WHERE avg_age IS NOT NULL
```

department_id	department_name	avg_age
5	Research	60.2500000000000000
4	Administration	64.6666666666666667

Table 2: Information of Department with Multiple Employees

(c) According to the question, the SQL code is provided below, and the result is presented as Table 3.

```sql
WITH participation_counts AS (
    SELECT
        pno, COUNT(essn) AS participant_count
    FROM public.works_on
    GROUP BY pno
),
ranked_participation AS (
    SELECT DISTINCT participant_count
    FROM participation_counts
    ORDER BY participant_count DESC
    LIMIT 1 OFFSET 1  -- get second most
)
SELECT
    pc.pno AS project_number,
        pname AS project_name,
    pc.participant_count
FROM participation_counts pc
JOIN
    ranked_participation rp ON pc.participant_count = rp.
       participant_count
```

```
LEFT JOIN public.project ON pno=pnumber
ORDER BY pc.participant_count DESC;
```

project_number	project_name	participant_count
1	ProductX	2
3	ProductZ	2

Table 3: Information of Projects with the second most Participants

(d) According to the question, the SQL code is provided below, and the result is presented as Table 4.

```
WITH participation_counts AS (
    SELECT pno,
        COUNT(essn) AS participant_count
    FROM public.works_on
    GROUP BY pno
),
ranked_participation AS (
    SELECT DISTINCT participant_count
    FROM participation_counts
    ORDER BY participant_count DESC
    LIMIT 1 OFFSET 1  -- second most participants
),
second_most_participation AS (
    SELECT pno
    FROM participation_counts
    WHERE
        participant_count = (SELECT participant_count FROM
            ranked_participation)
)

SELECT
    e.ssn,
    e.lname AS last_name,
    CASE
        WHEN e.salary >= 40000 THEN 'yes'
        ELSE 'no'
    END AS salary_above_40000,
    m.ssn AS manager_ssn,
    m.lname AS manager_last_name
FROM public.employee e
JOIN public.works_on w ON e.ssn = w.essn
JOIN second_most_participation sp ON w.pno = sp.pno
```

```
JOIN public.department d ON e.dno = d.dnumber
JOIN public.employee m ON d.mgr_ssn = m.ssn;  -- manager
   ssn
```

ssn	last_name	salary_above_40000	manager_ssn	manager_last_name
123456789	Smith	no	333445555	Wong
666884444	Narayan	no	333445555	Wong
453453453	English	no	333445555	Wong
333445555	Wong	yes	333445555	Wong

Table 4: Information of the-second-most-participant Projects Participants

(e) According to the question, the SQL code is provided below, and the result is presented as Table 5.

```
SELECT
    e.ssn,
    COALESCE(COUNT(DISTINCT wo.pno), 0) AS project_count,
    COALESCE(SUM(wo.hours), 0) AS total_hours,
    COALESCE(COUNT(DISTINCT p.plocation), 0) AS
       location_count
FROM public.employee e
LEFT JOIN
    public.works_on wo ON e.ssn = wo.essn AND wo.hours IS
       NOT NULL
LEFT JOIN
    public.project p ON wo.pno = p.pnumber
GROUP BY  e.ssn
ORDER BY  e.ssn;
```

ssn	project_count	total_hours	location_count
123456789	2	40.0	2
333445555	4	40.0	3
453453453	2	40.0	2
666884444	1	40.0	1
888665555	0	0	0
987654321	2	35.0	2
987987987	2	40.0	1
999887777	2	40.0	1

Table 5: Employee Participation Information

(f) According to the question, the SQL code is provided below, and the result is presented as Table 6.

```
WITH no_subordinate AS (
    SELECT e.ssn
    FROM public.employee e
    LEFT JOIN public.employee sub ON e.ssn = sub.super_ssn
    WHERE sub.ssn IS NULL
), project_participation AS (
    SELECT essn, COUNT(DISTINCT pno) AS project_count
    FROM public.works_on
    GROUP BY essn
)

SELECT ns.ssn, COALESCE(pp.project_count, 0) AS
   project_count
FROM no_subordinate ns
LEFT JOIN project_participation pp ON ns.ssn = pp.essn
ORDER BY ns.ssn;
```

ssn	project_count
123456789	2
453453453	2
666884444	1
987987987	2
999887777	2

Table 6: Employees Without Subordinates and Their Project Count

2 Question 2

(a) The Python code for setting up is provided below:

```
import numpy as np
import pandas as pd
import psycopg2
import duckdb
import matplotlib.pyplot as plt

with open('db_password.txt', 'r') as file:
    db_password = file.read().strip()

psql_conn = psycopg2.connect("dbname = 'Online Learning'
   user = 'postgres' host = 'localhost' password = " +
   db_password)
```

```
table_names = ['Subscriptions', 'StateChanges', '
   UserMissions', 'Answers']
con = duckdb.connect()

for table_name in table_names:
    query_str = "SELECT * FROM " + table_name
    df = pd.read_sql_query(query_str, psql_conn)
    con.register(table_name, df)

psql_conn.close()

# function that sees SQL command AS input and output table
    from database
def query(query):
    result = con.execute(query).fetchall()
    column_names = [desc[0] for desc in con.description]
    return column_names, result
```

According to the question, the Python code is provided below, and the result is presented as Table 7.

```
columns, data = query('''
          SELECT
            answers.answerid, answers.userid, answers.
               questionid, answers.missionid, answers.
               iscorrect, answers.costtime, answers.
               createdat, subscriptions.endedat
          FROM answers
          LEFT JOIN subscriptions ON answers.userid =
             subscriptions.subscriberid
          WHERE
            answers.createdat >= subscriptions.endedat
            AND answers.createdat >= '2021-05-01'
          ''')

# print out the first 5 rows of the data in latex
print(pd.DataFrame(data, columns=columns).head(5).to_latex
   (index=False))
```

(b) According to the question, the Python code is provided below, and the result is presented as Figure 1.

```
columns, data = query('''
```

answerid	userid	questionid	missionid	iscorrect	costtime	createdat	endedat
a3455948	u0004445	q0236673	NaN	1	5	2021-05-01 00:00:07	2021-03-02 23:59:59
a3455952	u0004445	q0236085	NaN	1	4	2021-05-01 00:00:18	2021-03-02 23:59:59
a3455954	u0004445	q0236032	NaN	1	8	2021-05-01 00:00:30	2021-03-02 23:59:59
a3455955	u0004445	q0260118	NaN	1	3	2021-05-01 00:00:39	2021-03-02 23:59:59
a3455958	u0004445	q0234539	NaN	1	3	2021-05-01 00:00:48	2021-03-02 23:59:59

Table 7: Sample Data of Questions Answered by Unsubscribed Users

```
            SELECT
                AVG(answers.costtime) AS avg_costtime, SUM
                   (answers.iscorrect) / COUNT(answers.
                   answerid) AS correct_rate
            FROM answers
            LEFT JOIN subscriptions ON answers.userid =
               subscriptions.subscriberid
            WHERE
                answers.createdat >= subscriptions.endedat
                AND answers.createdat >= '2021-05-01'
            GROUP BY answers.userid
            ''')
# plot scatter plot of average time spent on each question
    vs correct rate
df = pd.DataFrame(data, columns=columns)
plt.scatter(df['avg_costtime'], df['correct_rate'])
plt.xlabel('Average second spent on each question')
plt.ylabel('Correct rate')
# reverse x-axis
plt.gca().invert_xaxis()
# add median line
plt.axvline(x=df['avg_costtime'].median(), color='r',
   linestyle='--')
plt.axhline(y=df['correct_rate'].median(), color='r',
   linestyle='--')
plt.title('Average second spent on each question vs
   correct rate')
```

(c) The Python code for setting up is provided below:

```
import time
def get_table(table_name):
    columns, data = query('''
```

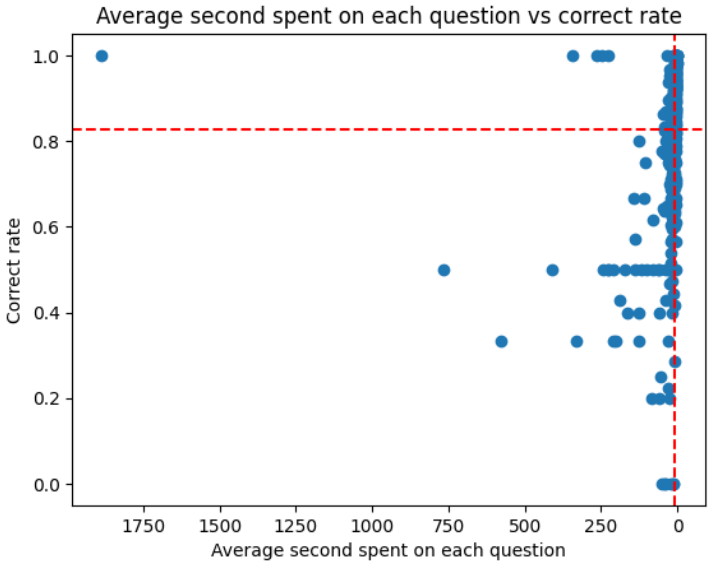

Figure 1: Average Second Spent on Each Question vs Correct Rate

```
                SELECT * FROM ''' + table_name)
    return pd.DataFrame(data, columns=columns)

answers = get_table('answers')
subscriptions = get_table('subscriptions')
```

The python code for option (i) is shown below:

```
start = time.time()
merged_data = pd.merge(answers, subscriptions, left_on='
   userid', right_on='subscriberid', how='left')
merged_data = merged_data[(merged_data['createdat_x'] >=
   merged_data['endedat']) & (merged_data['createdat_x']
   >= '2021-05-01')]
merged_data = merged_data[['answerid', 'userid', '
   questionid', 'missionid', 'iscorrect', 'costtime', '
   createdat_x', 'endedat']]
print(merged_data.head(5))
end = time.time()
print('Time taken to merge data for option (i): ', end -
   start)
```

And the python code for option (ii) is shown below:

```
start = time.time()
answers_filtered = answers[(answers['createdat'] >= '
   2021-05-01')]
```

```
merged_data = pd.merge(answers_filtered, subscriptions,
    left_on='userid', right_on='subscriberid', how='left')
merged_data = merged_data[(merged_data['createdat_x'] >=
    merged_data['endedat'])]
merged_data = merged_data[['answerid', 'userid', '
    questionid', 'missionid', 'iscorrect', 'costtime', '
    createdat_x', 'endedat']]
print(merged_data.head(5))
end = time.time()
print('Time taken to merge data for option (ii): ', end -
    start)
```

The output (ignoring printing the heads, since the result is identical with Table 7), which is the execution time is:

```
Time taken to merge data for option (i):
    1.9167506694793701
Time taken to merge data for option (ii):
    0.07049107551574707
```

Obviously, option (ii) is a faster solution. The reason why option (ii) is faster lies in the order of operations and the size of the data being processed at each step. In option (i), the code first merges the entire answers and subscriptions tables, resulting in a potentially large intermediate dataset that needs to be filtered afterwards. The merging operation, is computationally expensive and contributes to the longer execution time.

In contrast, option (ii) first filters the answers table to exclude rows, reducing the number of rows before performing the merge. By working with a smaller subset of data during the merge, the subsequent operations are faster and more efficient. This pre-filtering reduces the amount of data that needs to be processed, making each step faster. Thus, it is reasonable that option (ii) is faster because it minimizes the size of the data involved in the merge.

3 Question 3

(a) In my design, there will be three tables, and the corresponding relational schema will be shown in Figure 2.

In this design, there some columns needed to be explained:

- `is_visit`: Boolean value, 1 indicating that the employee is checking out for visiting customers.
- `attendance_id`: Unique string, when an employee check-in to work, an attendance_id will be created, while an attendance_id must require the employee to both check-in and check-out.

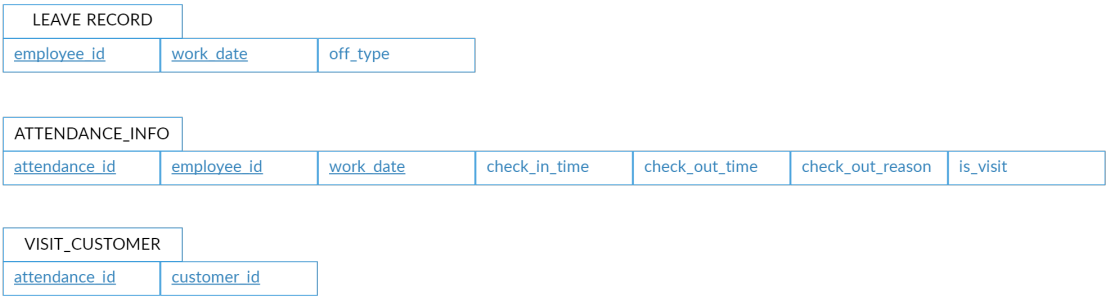

Figure 2: Redesigned Relational Schema for 3(a)

- `customer_id`: Unique string, representing a customer. (Note: it is not unique in table `VISIT_CUSTOMER`, but it should be unique in another entity, e.g. `CUSTOMER`)

The design fulfills the following levels of normal form:

(i) 1NF: The design is now in 1NF, since all multi-valued attributes are set to single-valued by creating tuples.

- `customer_id`: It is now in a new table, where duplicated `attendance_id` can map multiple customers as exactly one customer in a row if the employees visit more than one customer after one attendance.
- `check_in_time , check_out_time , check_out_reason`: Since one attendance corresponds to exactly one check-in and one check-out, there's no need to be represented as multiple-valued attributes.

(ii) 2NF: We can check whether its nonprime attributes are fully dependent on its candidate keys. It is in 2NF form.

- `LEAVE_RECORD`: {`employee_id , work_date`} is the only candidate key, where `leave_full` is fully dependent on the only candidate key.
- `ATTENDANCE_INFO`: `attendance_id` or {`employee_id , work_date`} are its candidate keys. No nonprime attributes partially depends on any of the candidate keys.
- `VISIT_CUSTOMER`: There are no nonprime attributes in this table.

(iii) 3NF: There are no transitive dependencies among the tables, so it indeed fit in 3NF.

(iv) BCNF: No prime attributes depending on attributes that do not form a superkey, thus it does fit in BCNF.

(v) 4NF: It fit in 4NF, besides, if we simply create multiple tuples to let the original fit in 1NF, it also does not contain any non-trivial multi-valued dependencies.

(b) The current relational schema is in 2NF. Since {`EngineerID , CustomerID , ConsultingDate`} is the only candidate key, every nonprime attribute fully depends on the candidate key.

However, `Fee` depends on `ServiceType` while itself depends on the candidate key {`EngineerID , CustomerID , ConsultingDate`}, so it is not in 3NF for this transitive dependency. It can be modified to as follows:

```
Consulting(EngineerID, CustomerID, ConsultingDate,
   ServiceType),
Service_Fee(ServiceType, Fee)
```

In this way, the transitive dependency can be removed and is thus fit in 3NF.

(c) The designed relational schema is shown in Figure 3.

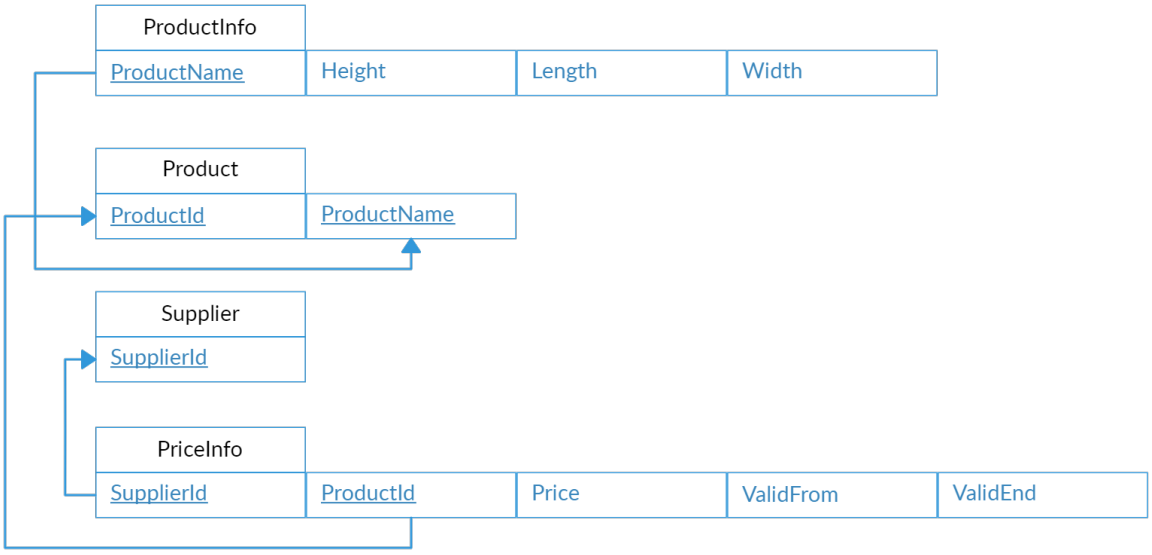

Figure 3: Redesigned Relational Schema for 3(b)

The design fulfills the following levels of normal form:

(i) 1NF: The design is now in 1NF, since all multi-valued attributes are set to single-valued by creating tuples. For example, in table `PriceInfo`, `ProductId` is converted to multiple rows in single value form.

(ii) 2NF: Every nonprime attributes are fully dependent on its candidate keys.

- `Product`: `ProductId` or `ProductName` are the candidate keys, where the rest are all fully dependent on each of the candidate keys.
- `PriceInfo`: {`SupplierId, ProductId, Price`} is the only candidate key. No nonprime attributes partially depends on any of the candidate keys.

(iii) 3NF: There are no transitive dependencies among the tables, so it indeed fit in 3NF. The adjustment made to fit in 3NF is to split `ProductInfo` out of `Product`, or else the information (e.g. height) was dependent on `ProductName` and itself is dependent to `ProductId`.

(iv) BCNF: No prime attributes depending on attributes that do not form a superkey, thus it does fit in BCNF.

(v) 4NF: It fit in 4NF. It does not contain any nontrivial multi-valued dependency.

4 Question 4

(a) Using SQL queries to interact with a database allows for precise control over the data retrieval process, providing the flexibility to write complex queries and optimize performance. However, raw SQL queries require developers to write database-specific code, which may lead to poor code maintenance and a higher risk of SQL injection attacks if not handled properly.

On the other hand, using an ORM (Object-Relational Mapping) abstracts away the underlying database interactions and allows developers to work with the database using the programming language's native objects. This can improve code readability and maintainability, especially in large applications. The downside is that ORMs may generate less optimized SQL queries and can have performance limitations for complex queries. Additionally, learning the ORM's conventions and configurations may add some initial overhead.

(b) A common database attack is Privilege Escalation, where an attacker exploits vulnerabilities in the database system or configuration to gain higher access privileges than originally permitted. This can happen through poorly configured user roles, unpatched vulnerabilities, or exploiting application flaws to escalate privileges.

To prevent privilege escalation, it's important to follow the principle of least privilege by granting users only the minimum access necessary for their tasks. Regularly updating and patching the database management system and applications can also help mitigate known vulnerabilities. Implementing proper logging and monitoring can detect unusual access patterns or unauthorized access attempts.

(c) "Separating front-end and back-end" is not exactly the same as "separating client-side and server-side." The front-end refers to the user interface and user experience components, while the back-end handles the "logic, involving database interactions, business logic, and server configuration.

"Separating client-side and server-side" refers to the architecture where the client-side executes code on the user's device (e.g., browser) and the server-side executes code on the server. In modern web development, the front-end may be partially rendered on the server-side for server-side rendering (SSR) or on the client-side for single-page applications (SPAs), which shows the subtle differences in these concepts.