OkCupid Database Design & Application

Candidate Number
82393, 21908, 31467, 22451

Contents

1.Introduction

This project aims to construct an efficient SQL database application to support in-depth analysis based on the data from the popular online dating app OkCupid. In our report, we have downloaded a comprehensive dataset from Kaggle, featuring detailed user profiles and preferences, and we also generated some synthetic data as a complement to our real datasets, to better demonstrate the completeness of this database function. This dataset will serve as the core of our database, enabling us to construct a multi-faceted picture of the user base.

2.Database Modeling

2.1 Mini-world description

The most important concept in a dating app is the user who has his or her own profile and personal preference.

The users can swipe other users and decide if they like them by swiping left(dislike) and right(like). If two users have both swiped each other right (like) , they become a match. The matched two can send messages to each other. After messaging, one user of the matched two can find the other user annoying and sending unpleasant photos, thus he/she can block him/her. By blocking the other person of the match, the blocked user cannot send the message to him/her.

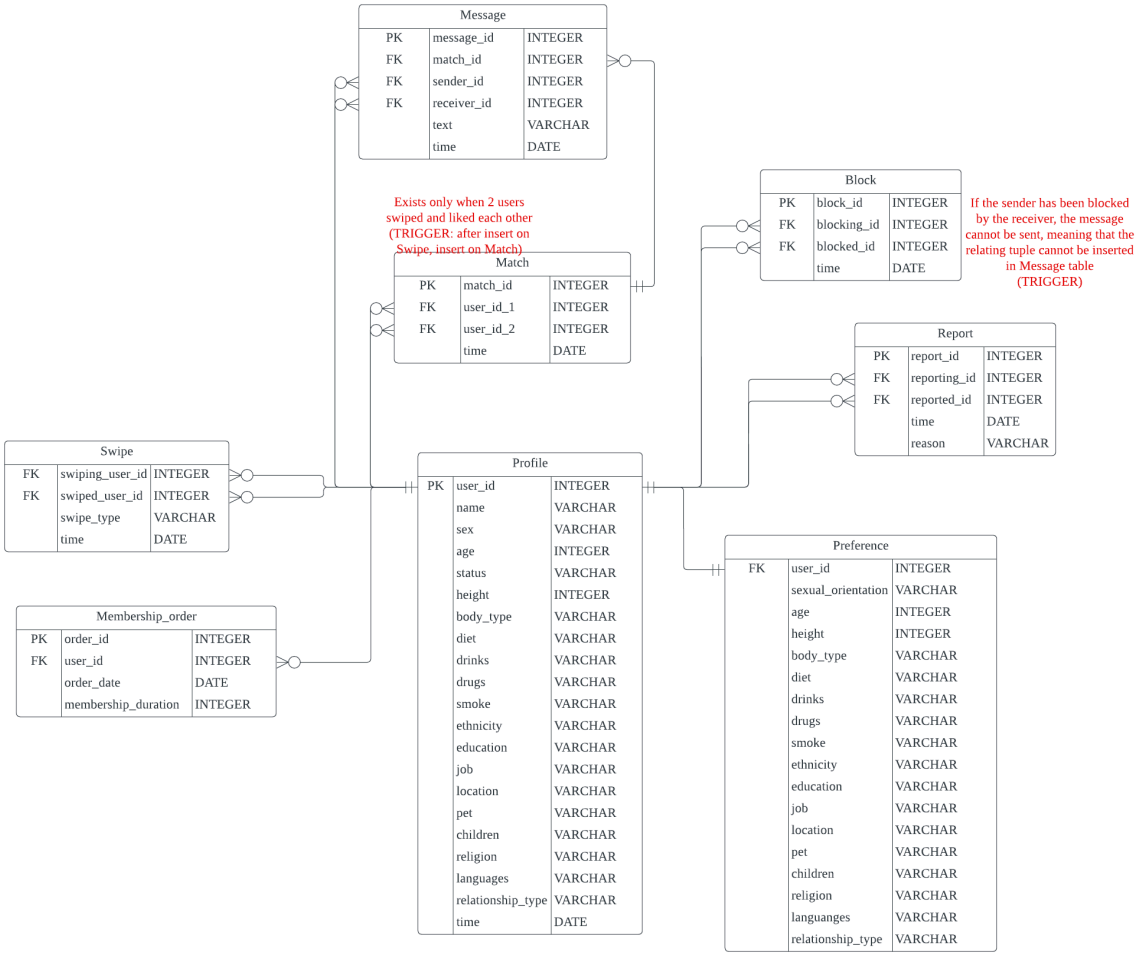
Report is another function that users can use. This report function is not limited to the matched two. For example, if one user finds another user is using fake information on the profile or there are pornographic content in the profile, the user can report that other user.

The users can purchase premium memberships for different durations. The premium memberships allow them to get access to more information, for instance, they can see who swiped them right (liked them).

2.2 ER diagram

This project developed a database program tailored for a dating app based on a real-world application, OKCupid. Because of the special characteristics of dating apps that most of the application actions are based on user profiles, our team decided to use a relational database which allows us to accurately map all the relationships between different entities and the essential profile entity. The entity relation diagram below indicates the structures and the tables in our relational model. The more detailed table creation commands and codes can be found in the interactive python notebook file uploaded.

[LucidChart]



2.3 Database description

Our team created a total of 8 tables in the database model to mimic the fundamental functions and situations of the dating apps: Profile, Preference, Report, Block, Swipe, Match, Message, Membership_order.

1) Profile, 2) Preference

The Profile and Preference tables are the most essential ones in the database model as most of the dating app operations are based on profiles and preferences. Profile table uses user_id as the Primary Key and has a series of attributes of the user. The Preference table shows the preferred partner features of the users; it has user_id as the Foreign Key that links the Preference table back to the Profile table. As the cardinality in the ER diagram shows, the Profile and Preference table are

one-to-one, which could be integrated into one table. However, for the simplicity and clarity of the database model, we distinguished two tables and created them separately.

3) Report
Users can report other users if they feel offended; it has Primary Key report_id and has reporting_id and reported_id as foreign keys which are linked back to the Profile table. The Foreign Key Constraint ensures that the reporting_id and reported_id must come from the existing user_id in the Profile Table.

4) Block
The Block table indicates the user can block other users if he/she no longer wants to receive messages from the other user; it has a similar structure as the Report table, with block_id as Primary Key and Foreign Key Constraint.

5) Swipe
The Swipe table shows the swiping situations between users, with two Foreign Keys swiping_user_id and swiped_user_id linking back to the Profile table, and attributes of swiping_type (left or right indicating dislike or like) as well as swiping time.

6) Match
The Match table describes the matching situation between users. The Match table has the Primary Key match_id and two Foreign Key user IDs linking back to the Profile table. The match between two users exists only when the two users have swiped each other and liked each other. The Match table is highly dependent on the Swipe table Therefore the Match table is not a direct table that we can populate data into. For the Match table, our team designed a TRIGGER after insert on swipe, which populates the corresponding IDs and time into the Match table when the two users have swiped and liked each other. The details and SQL code of the trigger will be addressed in the next section 2.4 Business Rules.

7) Message
The Message table shows the messages sent among users. The user can only send messages to the other user if they become a match. The table has a Primary Key message_id, a Foreign Key match_id linking back to the Match table, and two Foreign Key user IDs linking back to the Profile table.

8) Membership_order
The Membership_order table indicates the orders the users have made to upgrade to premium members. The premium members can unlock more functions, for instance, the members can see who swiped and liked her. In section 4 Database Usage, we will discuss in detail what functions the premium members have. This table has the Primary Key order_id and Foreign Key user_id linking back to the fundamental Profile table.

2.4 Business rules, constraints, triggers

In reality, there can be multiple constraints on the database based on business rules and the design of the application. For instance, as mentioned in the previous 2.3 Database description section, the match only exists when two users have swiped and liked each other. In order to mimic real-world scenarios, this project creates different constraints and triggers for the database for consistency and integrity purposes.

1) Match Trigger
This trigger says that if the two users have swiped and liked each other, then they will become a match and the corresponding data will be populated into the Match table.

```
CREATE TRIGGER matches AFTER INSERT ON Swipe
WHEN ( (SELECT time FROM Swipe
```

```
        WHERE swiping_user_id=NEW.swiped_user_id AND swiped_user_id=NEW.swiping_user_id AND
        swipe_type= 'right' AND NEW.swipe_type= 'right'
         )IS NOT NULL)
BEGIN
INSERT INTO Match(user_id_1,user_id_2,time) VALUES(NEW.swiped_user_id, NEW.swiping_user_id,
NEW.time);
END;
```

[Trigger Logic]: According to this trigger, after inserting a new tuple with user1 swiping user2 into Swipe, if the swipe type is like and there is an existing tuple that has the user2 liked user1, the data tuple will be inserted into the Match table with user1 and user2 IDs as well as the latest swipe time as the match time.

2) Block Trigger

This trigger says that when sending messages or inserting the tuples into the Message table the sender cannot send the message if the receiver has blocked the sender.

```
CREATE TRIGGER blocks BEFORE INSERT ON Message
BEGIN
        SELECT CASE
        WHEN ((SELECT block_id FROM Block
                WHERE blocking_id=NEW.receiver_id AND blocked_id=NEW.sender_id
                ) IS NOT NULL)
        THEN RAISE(FAIL, 'ERROR: Message cannot be sent. The sender has been blocked by the receiver.')
        END;
END;
```

[Trigger Logic]: Before inserting on the Message table if there exists a block tuple in the Block table that the NEW.receiver_id=blocking_id and NEW.sender_id=blocked_id, the message cannot be sent because the sender has been blocked by the receiver. The error message will say: Message cannot be sent. The sender has been blocked by the receiver.

3. Database Insertion

3.1 Data Description

The data of this project is a combination of real-world data and synthetic data. The real-world data part comes from the csv file we downloaded from Kaggle about OkCupid profiles. https://www.kaggle.com/datasets/andrewmvd/okcupid-profiles

As we can only acquire data of user profiles from Kaggles, more data such as user preferences and swipes need to be generated to complete the dating app database.

3.2 Synthetic Data Creation

The detailed synthetic data creation can be found in the uploaded interactive python notebook file. Via writing Python codes, we are able to generate random data in bulk. For each entity, a Python class is created not only to store the required attributes but also to randomly generate the desired data. In particular, numerical values are obtained using numpy.random.randint(), whereas random strings are chosen with numpy.random.choice(predefined_choices). Choices mainly come from the profile attributes data but with some change or reduction for convenience in later queries.

In addition to the simple random functions, we managed to assign probabilities to random.choice() based on the real-world scenarios and internet research. The detailed information can be found on the utils.py file, here is a picture of the distributions assigned.

```
#We particularly assigned the distribution to the options according to real-world scenarios and research
from the Internet
        preference = {
            "sexual_orientation": np.random.choice(["straight", "bisexual", "gay"],p=[0.9,0.08,0.02]),
            "age": np.random.randint(18, 55),
            "height": np.random.randint(59, 83),
            "body_type": np.random.choice(body_types,p=
[0.2,0.05,0.05,0.05,0.2,0.05,0.05,0.1,0.05,0.05,0.15]),
            "diet": np.random.choice(diet_update,p=[0.05,0.05,0.8,0.03,0.07]),
            "drinks": np.random.choice(drinks,p=[0.05,0.4,0.05,0.1,0.1,0.2,0.1]),
            "drugs" : np.random.choice(drugs,p=[0.1,0.1,0.3,0.5]),
            "smoke": np.random.choice(smokes,p=[0.3,0.4,0.1,0.05,0.05,0.1]),
            "ethnicity": np.random.choice(ethnicities_update,p=
[0.08,0.1,0.4,0.025,0.025,0.025,0.025,0.3,0.02]),
            'education' : np.random.choice(educations_update,p=[0.5,0.1,0.3,0.1]),
            'job' : np.random.choice(jobs),
            'location' : np.random.choice(locations_update,p=
[0.9,0.01,0.01,0.01,0.01,0.01,0.01,0.01,0.01,0.01,0.01]),
            'pet' : np.random.choice(pets_update),
            'children' : np.random.choice(children_update,p=[0.3,0.5,0.2]),
            'religion' : np.random.choice(religions_update,p=
[0.05,0.05,0.05,0.05,0.05,0.05,0.05,0.05,0.6]),
            'languages' : np.random.choice(languages_update,p=
[0.7,0.03,0.03,0.03,0.03,0.03,0.03,0.03,0.03,0.03,0.03]),
            'relationship_type' : np.random.choice(relationship_types)
        }
```

Also, in the synthetic data generation process, because of the nature and the design of the database, there are a lot of consistency constraints that we have to consider. For instance,considered that only the matched users can message each other, we have to use the random function of messaging users within the match table set (which is made possible by the combination of sql select and python for loop as the first pic below shows). The other detailed constraints can be found in the ipynb file.

```
In [18]: #Populating synthetic data of the Message table

         #According to the report section 2.1 Mini-world description, only the matched two can messag
         #We try to mimic and follow this rule in synthetic data generation
         #We read from the existing match table and limit the two receivers/senders to the matched tw

         c.execute("SELECT * FROM Match")
         result = c.fetchall()

         for match_id, uid1, uid2, start in result:
             start = datetime.datetime.strptime(start, '%Y-%m-%d %H:%M:%S')
             end = datetime.datetime(2023, 12, 31)
             reps = np.random.randint(0, 15)
             for i in range(reps):
                 rd = str(random_date(start, end))
                 text = np.random.choice(texts)
                 if i % 2 == 0:
                     message = Message(match_id, uid1, uid2, text, rd)
                 else:
                     message = Message(match_id, uid2, uid1, text, rd)
                 messages.append(message)

         for message in messages:
           d = message
           c.execute("INSERT INTO Message(match_id,sender_id,receiver_id,text,time) VALUES (?, ?, ?,
                     (d.ma_id,
                      d.u1_id,
                      d.u2_id,
                      d.text,
                      d.time)
           )
```

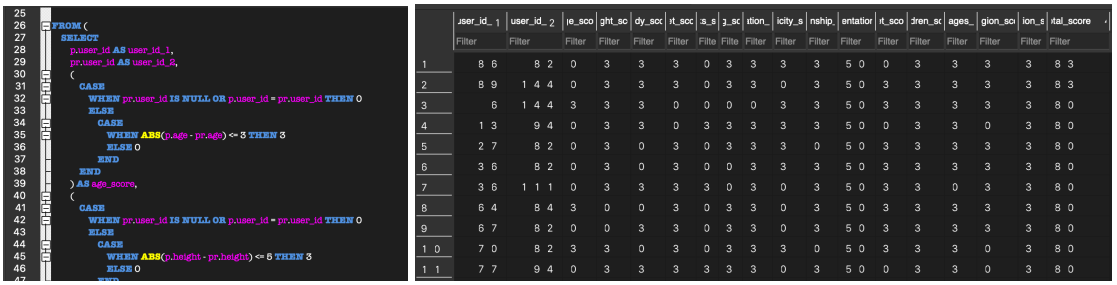
4. Database Usage

4.1 (Query1) **Match compatibility**

We figured out how well the preference of each user matches the others' profiles and then sort them to find the top users that best matches the user.

In order to find the most compatible other users for an individual user, we have to write a sql code to pre-computer the compatibility scores between each pair of users by comparing the profile tables of other users and the preference table of the individual user. Regarding the code for pre-computer compatibility scoring, the logic behind compatibility score calculation code is that: if the data is numerical, a range is set, and points are added if the profile data falls within this range; for data of character type, the SQL 'LIKE' statement is used to identify matching characters in the data, and points are added if there is a match.We turned the scores of every pair of users into a temporary table.The individual user 1, for example, can then query her compatibility scores with other users based on this temporary table. Therefore this Match compatibility is a two-step query.

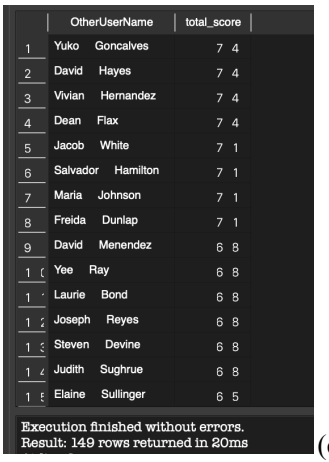
We have considered the dynamic changes and new users registration in designing this query. When there are new profiles data adding or updating to the database, we can simply rerun the compatibility calculation sql code to obtain an updated version of the temporary table. And then the individual user can obtain the updated compatibility information based on that updated temporary table.



[This is a part of the compatibility score calculation code, and part of the temporary table results, the complete code can be found in the Q1 sqlpro file uploaded]

```
SELECT
   (SELECT name FROM Profile WHERE user_id = tcs.user_id_2) AS OtherUserName,
   tcs.total_score
FROM
   TempCompatibilityScores tcs
WHERE
   tcs.user_id_1 = 1
ORDER BY 2 DESC;
```

[Query Logic]: This code allows the individual user to view the compatibility scores of him/her with other users. We used the case of user1, to see the compatibility between other users and user1.



	OtherUserName	total_score
1	Yuko Goncalves	74
2	David Hayes	74
3	Vivian Hernandez	74
4	Dean Flax	74
5	Jacob White	71
6	Salvador Hamilton	71
7	Maria Johnson	71
8	Freida Dunlap	71
9	David Menendez	68
10	Yee Ray	68
11	Laurie Bond	68
12	Joseph Reyes	68
13	Steven Devine	68
14	Judith Sughrue	68
15	Elaine Sullinger	65

Execution finished without errors.
Result: 149 rows returned in 20ms

(compatibility of other users and user 1)

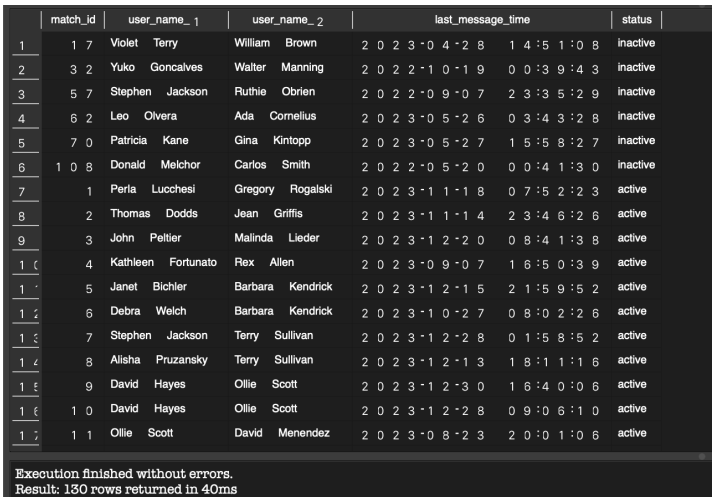
4.2 (Query2) **Inactive match**

For each match, we find the date of its latest message, check whether it's over 6 months ago and thus indicate the status(active/inactive).

```
SELECT m.match_id,
      p1.name AS user_name_1,
      p2.name AS user_name_2,
      MAX(me.time) AS last_message_time,
      CASE
         WHEN julianday('2024-01-03') - julianday(MAX(me.time)) > 182 THEN 'inactive'
         ELSE 'active'
      END AS status
FROM message me
JOIN match m ON me.match_id = m.match_id
JOIN profile p1 ON m.user_id_1 = p1.user_id
JOIN profile p2 ON m.user_id_2 = p2.user_id
GROUP BY me.match_id
ORDER BY CASE
          WHEN julianday('2024-01-03') - julianday(MAX(me.time)) > 182 THEN 1
          ELSE 2
       END,
       me.match_id;
```

[Query Logic]: First, we use the SELECT function to retrieve conversation IDs, user names, and timestamps of the last message. The MAX function identifies the most recent message in each

conversation. ’ julianday’ converts dates into numeric values to calculate the time elapsed since the last message, determining if a conversation is active or inactive. The ‘CASE’ statement categorizes conversations based on this activity. Then,the Join function is used to link messages with user profiles and conversation data. Finally,use ‘GROUP BY’ to organize the data by conversation, and the ORDER BY function sorts the results, prioritizing inactive conversations.



	match_id	user_name_1	user_name_2	last_message_time	status
1	17	Violet Terry	William Brown	2023-04-28 14:51:08	inactive
2	32	Yuko Goncalves	Walter Manning	2022-10-19 00:39:43	inactive
3	57	Stephen Jackson	Ruthie Obrien	2022-09-07 23:35:29	inactive
4	62	Leo Olvera	Ada Cornelius	2023-05-26 03:43:28	inactive
5	70	Patricia Kane	Gina Kintopp	2023-05-27 15:58:27	inactive
6	108	Donald Melchor	Carlos Smith	2022-05-20 00:41:30	inactive
7	1	Perla Lucchesi	Gregory Rogalski	2023-11-18 07:52:23	active
8	2	Thomas Dodds	Jean Griffis	2023-11-14 23:46:26	active
9	3	John Peltier	Malinda Lieder	2023-12-20 08:41:38	active
10	4	Kathleen Fortunato	Rex Allen	2023-09-07 16:50:39	active
11	5	Janet Bichler	Barbara Kendrick	2023-12-15 21:59:52	active
12	6	Debra Welch	Barbara Kendrick	2023-10-27 08:02:26	active
13	7	Stephen Jackson	Terry Sullivan	2023-12-28 01:58:52	active
14	8	Alisha Pruzansky	Terry Sullivan	2023-12-13 18:11:16	active
15	9	David Hayes	Ollie Scott	2023-12-30 16:40:06	active
16	10	David Hayes	Ollie Scott	2023-12-28 09:06:10	active
17	11	Ollie Scott	David Menendez	2023-08-23 20:01:06	active

Execution finished without errors.
Result: 130 rows returned in 40ms

[Screenshot of parts of the 130 results]

4.3 (Query3) **Most active match**

For every user who has at least one match with others, we find the person that the user has the maximum number of messages with.

```
SELECT user_id, user_name, most_messaged_user_id, most_messaged_user_name, MAX(message_count) AS
max_messages
FROM (
   SELECT p1.user_id AS user_id,
        p1.name AS user_name,
        p2.user_id AS most_messaged_user_id,
        p2.name AS most_messaged_user_name,
        COUNT(me.message_id) AS message_count
   FROM match m
   JOIN message me ON m.match_id = me.match_id
   JOIN profile p1 ON m.user_id_1 = p1.user_id
   JOIN profile p2 ON m.user_id_2 = p2.user_id
   GROUP BY m.user_id_1, m.user_id_2
   UNION ALL
   SELECT p1.user_id AS user_id,
        p1.name AS user_name,
        p2.user_id AS most_messaged_user_id,
        p2.name AS most_messaged_user_name,
        COUNT(me.message_id) AS message_count
   FROM match m
   JOIN message me ON m.match_id = me.match_id
   JOIN profile p1 ON m.user_id_2 = p1.user_id
   JOIN profile p2 ON m.user_id_1 = p2.user_id
   GROUP BY m.user_id_2, m.user_id_1
) AS combined
GROUP BY user_id
ORDER BY user_id;
```

[Query Logic]:This code involves two subqueries that count messages exchanged between user pairs, considering each user both as a sender and receiver. These subqueries JOIN user profiles with messages and are combined using the UNION ALL function to include all possible interactions. The outer query then selects their own userid and name, the userid and name of their most messaged contact, and the maximum number of messages exchanged with that contact. We GROUP BY the results by userid and order them accordingly (in order to show the inactive messages more clearly).

	user_id	user_name	most_messaged_user_id	most_messaged_user_name	max_messages
1	0	Perla Lucchesi	111	Laurie Bond	14
2	1	Gregory Rogalski	123	Evelyn Goodman	7
3	2	Thomas Dodds	10	Jean Griffis	13
4	3	Carlos Hamby	108	Jeffrey Degen	7
5	4	Susan Tshudy	108	Jeffrey Degen	1
6	5	Rhonda Carter	109	Bobbie Mccowan	2
7	6	John Peltier	75	Justin Rosado	9
8	7	Leo Olvera	126	Paul Quintero	11
9	8	Alisha Pruzansky	37	Terry Sullivan	12
10	9	Ernest Clark	74	Patricia Kane	14
11	10	Jean Griffis	2	Thomas Dodds	13
12	11	Travis Stamp	110	William Kocieda	9
13	12	Yuko Goncalves	71	Walter Manning	6
14	13	Jacob Harrington	53	Robert Sousa	6
15	14	Cathy Burns	60	Lee Owens	9
16	16	Toby Luna	45	David Menendez	12
17	17	Candice Phillips	121	Mary Wiedmann	14
18	18	Katrina Albanese	105	Carla Maltas	8

Execution finished without errors.
Result: 120 rows returned in 42ms

[Screenshot of parts of the results]

4.4 (Query4) **Membership function: who liked him/her**

For all the membership users, they are able to see who liked/ swiped him or her right.

```
SELECT
   (SELECT p.name FROM Profile p WHERE p.user_id = mo.user_id) AS VIP,
   GROUP_CONCAT((SELECT p.name FROM Profile p WHERE p.user_id = s.swiping_user_id)) AS Likes_you
FROM
   Membership_order mo
JOIN
   Swipe s ON mo.user_id = s.swiped_user_id
WHERE
   date(mo.order_date, '+' || mo.membership_duration || ' days') > '2024-01-01'
   AND s.swipe_type = 'right'
GROUP BY
   mo.user_id
ORDER BY
    VIP ASC;
```

[Query Logic] :I use this SQL query to extract the names of VIP members from a dating app, along with a concatenated list of users who have liked each VIP. I combine membership and swipe data, filtering out inactive memberships before a specific date, and consider only right swipes. The results are grouped by each VIP's user ID and sorted by their names.

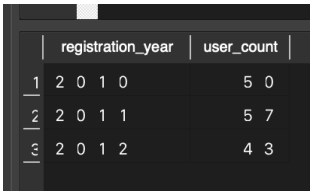
	VIP	Likes_you
1	Andy Larue	Miriam Pak,David Menendez,Violet Terry,Yee Ray,Carlos Donald,Lucy Ramirez,Carmen Tooley,Annette Schultz,Patsy Hinzman,Martha Books,William Kocieda,Mary Wiedmann,James Hayes,Robert Hankins
2	Barbara Kendrick	Travis Stamp,Janet Bichler,Debra Welch,Delia Maurer,Melissa Yale,David Menendez,Carlos Donald,Whitney Johnson,Stella Solis,Cheryl Beagle,Barry Mcdonald,Walter Arroyo,Vivian Hernandez,Eloise Dean,Judy Tesh,Billy Shary,Hazel Knox,Judith Sughrue,John Jones
3	David Hayes	Leo Olvera,Janet Bichler,Rex Allen,Miriam Pak,Ollie Scott,Gloria Vandenbosch,Maria Johnson,Freida Dunlap,Barry Mcdonald,Amanda Parker,Jack Mcneil,William Kocieda,Hazel Knox,Mary Wiedmann,Kevin Catton,Robert Hankins
4	David Mercer	John Peltier,Travis Stamp,Katrina Albanese,Janis Hartman,Violet Terry,Edna Clayton,Robert Milian,Susan English,Amanda Parker,Judy Tesh,Jesus Brown,Jeffrey Degen,Orlando Oberst,Marie White,James Hayes,Robert Hankins
5	Elaine Sullinger	Gregory Rogalski,Jerry Holsinger,Malinda Lieder,David Menendez,Derrick Gean,Gina Belt,Stella Solis,Freida Dunlap,Martha Books,Paul Quintero,Fredrick Garcia,George Brown,James Hayes,Steven Devine
6	Eloise Dean	Travis Stamp,Jose Taylor,Latonya Manigault,Candace Lytle,Susan English,Freida Dunlap,Sharon Abbott,Ada Cornelius,Gina Kintopp,Billy Shary,Ricky White,Steven Devine
7	Ernest Clark	Janis Hartman,Terry Sullivan,Ollie Scott,Jeanice Bowden,Derrick Gean,Jose Taylor,Lucy Ramirez,Patricia Kane,Robert Milian,Cheryl Beagle,Barry Mcdonald,Orlando Oberst,Joseph Reyes,Donald Melchor,Katie Mckinney,Carlos Smith,Jordan Bowen,Kevin Catton
8	Evelyn Goodman	Gregory Rogalski,Travis Stamp,David Hayes,Violet Terry,Derrick Gean,Jason Woodard,Gloria Vandenbosch,Maria Johnson,Freida Dunlap,Jack Mcneil,Steve High
		Perla Lucchesi,Susan Tshudy,Peter Ballintyn,Mary Price,Stella Solis,Carey Gary,Candace Lytle,Margaret

4.5 (Query5) **Trend analysis for dynamic preference for all users**

For profiles registered in each year, we find the percentage of every choice of attributes and find whether there is a trend. This is a series of queries with similar basic logics.

```
SELECT
   strftime('%Y', time) AS registration_year,
   COUNT(user_id) AS user_count
FROM
   Profile
GROUP BY
   registration_year
ORDER BY
   registration_year;
```

[Query Logic]: This SQL query extracts and counts the number of users who registered each year from Profile. I use the strftime('%Y', time) function to format the date data, specifically extracting the year from the 'time' column and labeling this as 'registration_year'. The query then groups the data by these extracted years and counts the number of user registrations in each year using COUNT(user_id). Finally, we order the results by the registration year in ascending order, providing a clear, year-wise distribution of user registrations.

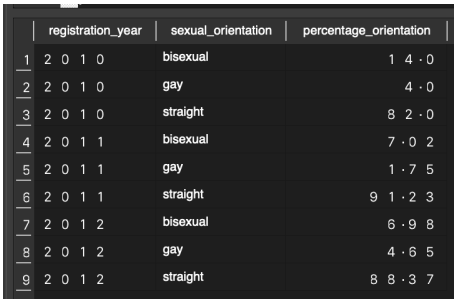


	registration_year	user_count
1	2010	50
2	2011	57
3	2012	43

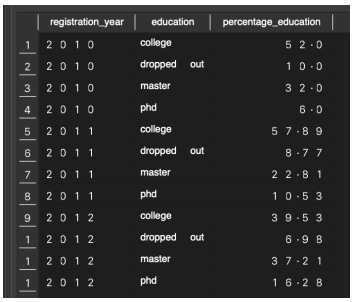
In order to implement this function, we need to analyze the changes in preferences of users with different registration years from multiple angles, so we selected eight different attributes: sexual orientation, education preference, age preference, child preference, pet preference, and career preference. , lifestyle preferences, language preferences. Therefore, next I will choose the SQL code of how to get the "child preference" data as an example to explain my code logic. I will directly show the results for other attributes, and the complete code will be presented in the appendix sqlpro file.

```
SELECT
   strftime('%Y', Profile.time) AS registration_year,
   CASE
      WHEN Preference.children ='want kids' THEN 'Want Kids'
      WHEN Preference.children = 'do not want kids' THEN 'Do not Want Kids'
      WHEN Preference.children = 'nan' THEN 'nan'
      ELSE 'Other Preferences'
   END AS children_preference,
   ROUND(COUNT(*) * 100.0 / SUM(COUNT(*)) OVER (PARTITION BY strftime('%Y', Profile.time)), 2) AS
percentage
FROM
   Profile
JOIN
   Preference ON Profile.user_id = Preference.user_id
GROUP BY
   registration_year, children_preference
ORDER BY
   registration_year, children_preference;
```

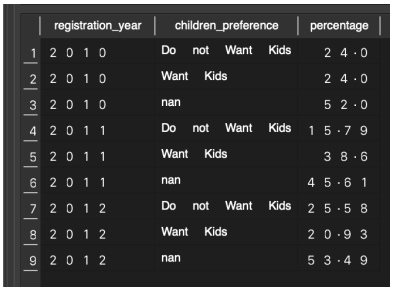
[Query Logic]: This SQL query joins two tables, 'Profile' and 'Preference', to analyze users' preferences about having children, categorized by the year they registered. It first extracts the registration year from the 'Profile' table using strftime('%Y', Profile.time). Then, a CASE statement categorizes the 'children' column from the 'Preference' table into 'Want Kids', 'Do Not Want Kids', 'nan', or 'Other Preferences'. The query calculates the percentage of each preference category per year using ROUND(COUNT(*) * 100.0 / SUM(COUNT(*)) OVER (PARTITION BY strftime('%Y', Profile.time)), 2). This gives the proportion of each preference in a given year. The results are grouped by registration year and children preference, and then ordered by the registration year and the children preference category.



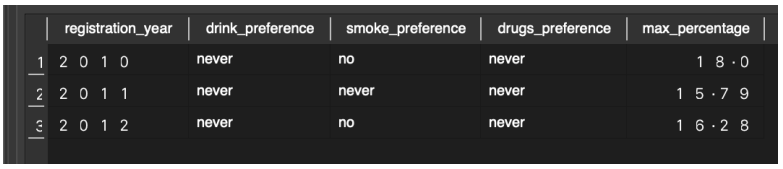
	registration_year	sexual_orientation	percentage_orientation
1	2010	bisexual	14.0
2	2010	gay	4.0
3	2010	straight	82.0
4	2011	bisexual	7.02
5	2011	gay	1.75
6	2011	straight	91.23
7	2012	bisexual	6.98
8	2012	gay	4.65
9	2012	straight	88.37



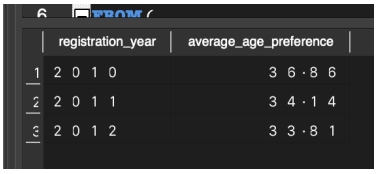
	registration_year	education	percentage_education
1	2010	college	52.0
2	2010	dropped out	10.0
3	2010	master	32.0
4	2010	phd	6.0
5	2011	college	57.89
6	2011	dropped out	8.77
7	2011	master	22.81
8	2011	phd	10.53
9	2012	college	39.53
10	2012	dropped out	6.98
11	2012	master	37.21
12	2012	phd	16.28



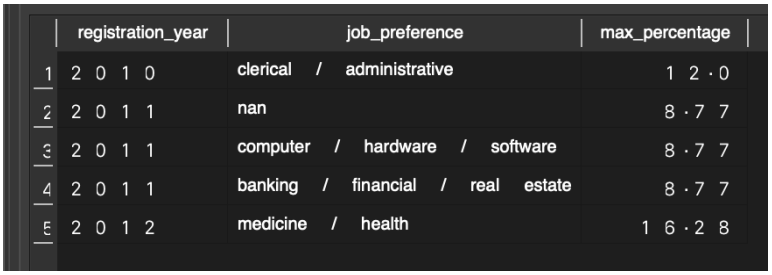
	registration_year	children_preference	percentage
1	2010	Do not Want Kids	24.0
2	2010	Want Kids	24.0
3	2010	nan	52.0
4	2011	Do not Want Kids	15.79
5	2011	Want Kids	38.6
6	2011	nan	45.61
7	2012	Do not Want Kids	25.58
8	2012	Want Kids	20.93
9	2012	nan	53.49



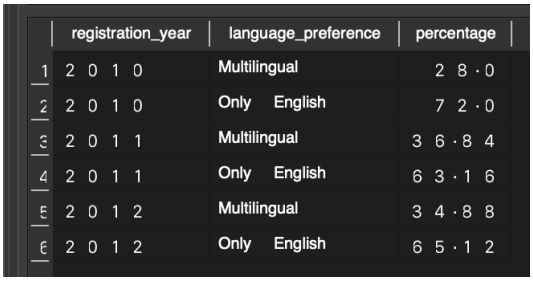
	registration_year	drink_preference	smoke_preference	drugs_preference	max_percentage
1	2010	never	no	never	18.0
2	2011	never	never	never	15.79
3	2012	never	no	never	16.28



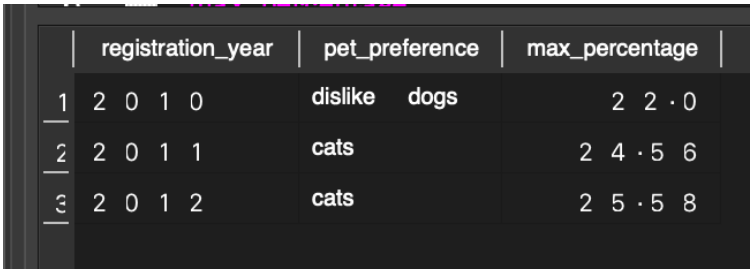
	registration_year	average_age_preference
1	2010	36.86
2	2011	34.14
3	2012	33.81



	registration_year	job_preference	max_percentage
1	2010	clerical / administrative	12.0
2	2011	nan	8.77
3	2011	computer / hardware / software	8.77
4	2011	banking / financial / real estate	8.77
5	2012	medicine / health	16.28



	registration_year	language_preference	percentage
1	2010	Multilingual	28.0
2	2010	Only English	72.0
3	2011	Multilingual	36.84
4	2011	Only English	63.16
5	2012	Multilingual	34.88
6	2012	Only English	65.12



	registration_year	pet_preference	max_percentage
1	2010	dislike dogs	22.0
2	2011	cats	24.56
3	2012	cats	25.58

(above are the results for the 8 preferences analysis; because of the page limit, not all the sql codes are shown; but we uploaded the detailed sqlpro files containing all the codes)

6. Database Application Technology

For our dating app database application, we used several technologies throughout the project. We decided to use sqlite library in Python because we believe the relational database can best represent the relationships of all the entities of the dating app and we choose sqlite out of all dbms specifically because it is a very light and flexible solution that caters our needs. Besides, we also used pandas and numpy libraries when inserting both real data and synthetic data. Pandas is for reading the Kaggle downloaded csv file, and Numpy is for multiple random functions that we used for producing synthetic data. In the Section5 Queries part, we used DB Browser for SQLite as a complement, directly running the sql queries in it, as DB Browser has a great representation of the database with structured tables and organized data presentation.

7. Conclusion

Our team designed a practical dating app database application in this project. We started by database modeling and considered different business constraints etc. After that, our team tried to insert all the data including real data downloaded as well as synthetic data that we generated mimicking the real world. Apart from building the database application itself, the project also focuses on the database usage and provides a series of queries related to the dating app database program. These interesting queries allow users to gain a deep and profound insight into the database as well as the data. In conclusion, this project provided us with a great opportunity to integratedly apply all the knowledge of this course, from the data modeling to the final queries, and we really enjoyed the process of turning the theoretical design and concepts into a real database program.