Editorial 602: Friend Requests

Problem Statement

Find people who have the most friends and the count of their friends.
Ref: https://leetcode.com/problems/friend-requests-ii-who-has-the-most-friends/description/

Sample Input: Table RequestAccepted with columns: requester_id (int), accepter_id (int), accept_date (date). Primary key is represented as PK in the below table.

requester_id (PK)	accepter_id (PK)	accept_date
1	2	'2016/06/03'
1	3	'2016/06/08'
2	3	'2016/06/08'
3	4	'2016/06/09'

Sample Output: Table with columns: id (int), count (int) representing the person id with most friends and the count of their friends.

id	count
3	3

Note: All solutions account for scenarios in which multiple individuals may have the same highest friend count in the test cases.

Solution(s)

There are multiple ways to solve this problem. Below are four approaches:

Approach 1 (Concat & Max Python Pandas)

This approach uses python pandas library to concatenate requester_id and accepter_id columns, counting number of friends for each person, and finding persons with maximum friends count.

Algorithm

1. Concatenate **requester_id** & **accepter_id** columns into a single column (**all_people**)
2. Use **value_counts** to count the occurrences of each person in **all_people**.
3. Find the maximum count using **max()**.
4. Filter the persons with the maximum count using boolean indexing.
5. Create a DataFrame with the result containing all persons with the maximum count.

Code

```
# Python Code

import pandas as pd

# Sample data
data = {'requester_id': [1, 1, 2, 3, 4, 5],
        'accepter_id': [2, 3, 3, 4, 1, 4],
        'accept_date': ['2016/06/03', '2016/06/08', '2016/06/08', '2016/06/09','2016/06/09',
'2016/06/09' ]}

df = pd.DataFrame(data)

# Combine requester_id and accepter_id into a single column
all_people = pd.concat([df['requester_id'], df['accepter_id']])

# Count occurrences of each person
friend_counts = all_people.value_counts()

# Find all persons with the maximum count
max_friends_ids = friend_counts[friend_counts == friend_counts.max()].index.tolist()
```

```
max_friends_count = friend_counts.max()

# Create the result DataFrame
result_df = pd.DataFrame({'id': max_friends_ids, 'count': [max_friends_count] *
len(max_friends_ids)})

# Print the result
print(result_df)
```

Time and Space Complexity

Time Complexity: O(n)

- Combining columns: O(n)
- Counting occurrences: O(n)
- Finding max: O(1) (as it's a single operation)
- Filtering: O(n)

Overall, the time complexity is O(n).

Space Complexity: O(n + m)

- Additional DataFrame ‘all_people’: O(n)
- Series ‘friend_counts’: O(n)
- Result DataFrame: O(m), where m is the number of persons with the maximum count.

Overall, the space complexity is O(n + m).

Edge case

An edge case would be when there are no friend connections in the input data (empty DataFrame). Example: df = pd.DataFrame({'requester_id': [], 'accepter_id': [], 'accept_date': []})

Approach 2 (Concat & Rank Python Pandas)

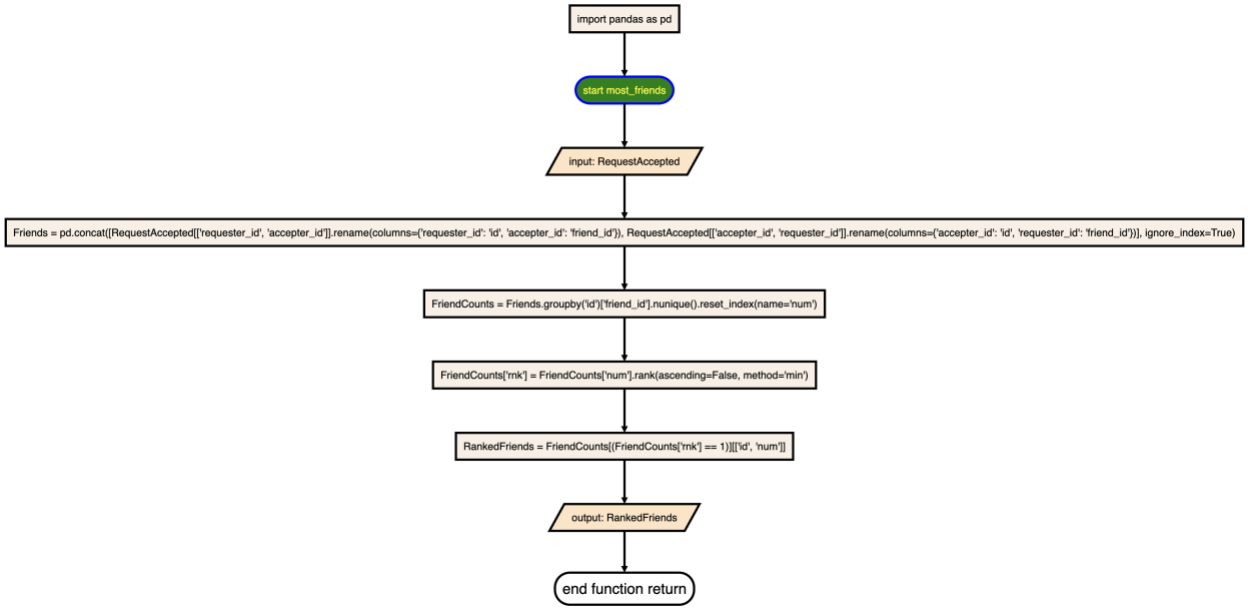
This approach uses python pandas library to concatenate, group by ID’s and count the number of friends for each person. Finally, the data is ranked to identify persons with the most friends.

Algorithm

1. Concatenate two DataFrames, **'requester_id'** and **'accepter_id'**, renaming columns to **'id'** and **'friend_id'**.
2. Count unique friend_ids with **nunique** for each person (grouping by id).
3. **Rank** people based on the number of friends, using the **'min'** method for tie breaking.
4. Select the **top-ranked** people (peoplewith the highest number of friends).

Code

```python
# Python Code

import pandas as pd

def most_friends(RequestAccepted):
    # Concatenate 'requester_id' and 'accepter_id' columns, rename for uniformity, and create a new DataFrame 'Friends'
    Friends = pd.concat([RequestAccepted[['requester_id', 'accepter_id']].rename(
        columns={'requester_id': 'id', 'accepter_id': 'friend_id'}),
                         RequestAccepted[['accepter_id', 'requester_id']].rename(
                             columns={'accepter_id': 'id', 'requester_id': 'friend_id'})],
                        ignore_index=True)

    # Group 'Friends' DataFrame by 'id', count unique 'friend_id' for each person, and reset index to form 'FriendCounts'
    FriendCounts = Friends.groupby('id')['friend_id'].nunique().reset_index(name='num')

    # Add a new column 'rnk' to 'FriendCounts' representing the rank of each person based on friend count
    FriendCounts['rnk'] = FriendCounts['num'].rank(ascending=False, method='min')

    # Filter 'FriendCounts' to include only individuals with the highest friend count and select 'id' and 'num' columns
    RankedFriends = FriendCounts[FriendCounts['rnk'] == 1][['id', 'num']]

    # Return the DataFrame containing individuals with the most friends and their friend count
    return RankedFriends
```



Time and Space Complexity

Time (run-time) Complexity: O(nlogn) (dominated by groupby and rank operations)

- Concatenation: O(n), where n is the number of rows in the input DataFrame.
- GroupBy and unique count: O(n log n), where n is the number of unique users.
- Ranking: O(n log n), where n is the number of unique users.
- Selection: O(n), where n is the number of unique users.

Space (Memory) Complexity: O(n) (due to intermediate DataFrames creation)

Edge case

Empty DataFrame (or) DataFrame with Null values

Approach 3 (Union All & Rank SQL)

This approach uses SQL queries to combine requester_id and accepter_id, counting distinct friends, ranking based on friend count, and filtering for individuals with the highest rank.

Algorithm

1. Create a temporary table **(Friends)** by combining requester_id and accepter_id.
2. Count distinct friends **(FriendCounts)** for each individual.

3. Rank individuals (**RankedFriends**) based on friend count in descending order.
4. Filter individuals with the highest rank (rnk =1).
5. Retrieve id and num columns from the filtered result.

Code

```mysql
/* MySQL Code */

-- Select 'id' and 'num' columns from the result set
SELECT id, num
FROM (
   -- Apply ranking based on the friend count in descending order and add a new column 'rnk'
   SELECT id, num, RANK() OVER (ORDER BY num DESC) AS rnk
   FROM (
      -- Count the number of distinct friends ('num') for each 'id' in the temporary 'Friends' table
      SELECT id, COUNT(DISTINCT friend_id) AS num
      FROM (
         -- Create a temporary table 'Friends' by combining 'requester_id' and 'accepter_id'
         SELECT requester_id AS id, accepter_id AS friend_id
         FROM RequestAccepted
         UNION ALL
         SELECT accepter_id AS id, requester_id AS friend_id
         FROM RequestAccepted
      ) AS Friends
      -- Group the temporary table by 'id'
      GROUP BY id
   ) AS FriendCounts
) AS RankedFriends
-- Filter the result to include only individuals with the highest friend count (rank = 1)
WHERE rnk = 1;
```

Time and Space Complexity

Time: O(nlogn)

- Subquery 1 (Friends): O(n) - Iterating through the RequestAccepted table once to create the temporary Friends table.
- Subquery 2 (FriendCounts): O(n) - Counting distinct friends for each id in the temporary Friends table.
- Subquery 3 (RankedFriends): O(n log n) - Applying the RANK() function over the result of FriendCounts.

Space: O(n), where n is the space required to store temporary tables.

Edge case(s)

1. RequestAccepted table is empty (or) not in the accepted input format.
2. Handle case where Rank() can be replaced for improved performance.
3. Handle the case where indexing and database optimization is involved.

Approach 4 (Sort List of Dictionaries Python)

This approach involves iterating through the request_accepted data to count the number of friends for each individual, then ranking individuals based on friend count in descending order. Finally, it filters individuals with the highest friend count and returns the result.

Algorithm

1. Initialize an empty dictionary **friend_count** to store the count of friends for each person.
2. Iterate through each dictionary in the **request_accepted** list.
3. For each dictionary, increment the friend count for both 'requester_id' and 'accepter_id'.
4. Sort the **friend_count** dictionary items in descending order based on the friend count.
5. Retrieve the highest friend count from the sorted list.
6. Return list of dictionaries (**result)**, with 'id', 'num' for persons with highest friend count.

Code

```
# Python Code

def find_most_friends(request_accepted):
    friend_count = {}

    for row in request_accepted:
        friend_count[row['requester_id']] = friend_count.get(row['requester_id'], 0) + 1
        friend_count[row['accepter_id']] = friend_count.get(row['accepter_id'], 0) + 1

    # Rank individuals based on the number of friends in descending order
    ranked_friends = sorted(friend_count.items(), key=lambda x: x[1], reverse=True)

    # Find the highest friend count
    max_friends = ranked_friends[0][1]

    # Filter individuals with the highest friend count
    result = [{'id': k, 'num': v} for k, v in ranked_friends if v == max_friends]

    return result

if __name__ == '__main__':
    request_accepted = [
        {'requester_id': 1, 'accepter_id': 2, 'accept_date': '2016/06/03'},
```

```
    {'requester_id': 1, 'accepter_id': 3, 'accept_date': '2016/06/08'},
    {'requester_id': 2, 'accepter_id': 3, 'accept_date': '2016/06/08'},
    {'requester_id': 3, 'accepter_id': 4, 'accept_date': '2016/06/09'}
]
print(find_most_friends(request_accepted))
```

Time and Space Complexity

Time: O(nlogn), where n is the number of unique individuals.

The code iterates through each entry in the request_accepted list once, performing constant time operations for each entry. Sorting the friend_count dictionary takes O(n log n) time.

Space: O(n), where n is the number of unique individuals.

Edge case

An empty list of 'request_accepted' or a list of dictionaries not adhering to the expected format.