**Facebook Advertiser Status**

**Key Concepts**

* Conditional update with join.
* Left join vs. inner join.
* Edge case: adding new users.
* Boolean algebra and simplification.

**Tables:**

* *DailyPay*: user\_id (showing today paid ads fee) on day T. **Only** advertisers who paid will show up in this table.
* *Advertiser*: two columns, user\_id and their status on day T-1 Use today’s payment log in *DailyPay* table to update status in *Advertiser* table

**Status:**

* New: users registered on day T.
* Existing: users who paid on day T-1 and on day T.
* Churn: users who paid on day T-1 but not on day T.
* Resurrect: users who did not pay on T-1 but paid on day T.

**State Transition**

[Diagram

Description automatically generated](https://github.com/NIteshx2/AdvancedSQL_Interview/blob/master/Interview/01_Facebook_Advertiser_Status/fig/transition.png)

| **#** | **Start** | **End** | **Condition** |
| --- | --- | --- | --- |
| 1 | NEW | EXISTING | Paid on day T |
| 2 | NEW | CHURN | No pay on day T |
| 3 | EXISTING | EXISTING | Paid on day T |
| 4 | EXISTING | CHURN | No pay on day T |
| 5 | CHURN | RESURRECT | Paid on day T |
| 6 | CHURN | CHURN | No pay on day T |
| 7 | RESURRECT | EXISTING | Paid on day T |
| 8 | RESURRECT | CHURN | No pay on day T |

By examining the above table. We can see that as long as user has not paid on day T, his status is updated to CHURN regardless of previous status (check with interviewer that all new users who registered on day T did pay, and if they didn't, they are not immediately considered as CHURN.

When user did pay on day T (#1, 3, 5, 7). They can become either EXISTING or RESURRECT, depending on their previous state. RESURRECT is only possible when previous state is CHURN. When previous state is anything else, status is updated to EXISTING.

**Sample Database**

Load the database file [db.sql](https://github.com/NIteshx2/AdvancedSQL_Interview/blob/master/Interview/01_Facebook_Advertiser_Status/db.sql) to localhost MySQL. An Advertiser database will be created with two tables.

mysql < db.sql -uroot -p

mysql> SELECT \* FROM Advertiser;

+----+---------+-----------+

| id | user\_id | status |

+----+---------+-----------+

| 1 | bing | NEW |

| 2 | yahoo | NEW |

| 3 | alibaba | EXISTING |

| 4 | baidu | EXISTING |

| 5 | target | CHURN |

| 6 | tesla | CHURN |

| 7 | morgan | RESURRECT |

| 8 | chase | RESURRECT |

+----+---------+-----------+

8 rows in set (0.00 sec)

mysql> SELECT \* FROM DailyPay;

+----+---------+------+

| id | user\_id | paid |

+----+---------+------+

| 1 | yahoo | 45 |

| 2 | alibaba | 100 |

| 3 | target | 13 |

| 4 | morgan | 600 |

| 5 | fitdata | 1 |

+----+---------+------+

5 rows in set (0.00 sec)

**Solution**

**Step 1. Update Existing Advertiser**

After simplifying the boolean algebra, we only need three conditions. State **explicitly** we don't need "ELSE status" in the CASE statement because we've covered all possible conditions. Also emphasize we need **LEFT JOIN** to find out who did not pay on day T.

UPDATE Advertiser AS a

LEFT JOIN DailyPay AS d

ON a.user\_id = d.user\_id

SET a.status = CASE

WHEN d.paid IS NULL THEN "CHURN"

WHEN a.status = "CHURN" AND d.paid IS NOT NULL THEN "RESURRECT"

WHEN a.status != "CHURN" AND d.paid IS NOT NULL THEN "EXISTING"

END;

Check the *Advertiser* to see if the update make sense.

mysql> SELECT \* FROM Advertiser;

+----+---------+-----------+

| id | user\_id | status |

+----+---------+-----------+

| 1 | bing | CHURN |

| 2 | yahoo | EXISTING |

| 3 | alibaba | EXISTING |

| 4 | baidu | CHURN |

| 5 | target | RESURRECT |

| 6 | tesla | CHURN |

| 7 | morgan | EXISTING |

| 8 | chase | CHURN |

+----+---------+-----------+

8 rows in set (0.00 sec)

**Step 2. Insert New Advertiser**

Note that we missed the new user. To find the new user, left join *DailyPay* with *Advertiser*. If there is no match on the right, the user is new.

INSERT INTO

Advertiser (user\_id, status)

SELECT d.user\_id

,"NEW" as status

FROM DailyPay AS d

LEFT JOIN Advertiser AS a

ON d.user\_id = a.user\_id

WHERE a.user\_id IS NULL;

Check again that the new users are added.

SELECT \* FROM Advertiser;

+----+---------+-----------+

| id | user\_id | status |

+----+---------+-----------+

| 1 | bing | CHURN |

| 2 | yahoo | EXISTING |

| 3 | alibaba | EXISTING |

| 4 | baidu | CHURN |

| 5 | target | RESURRECT |

| 6 | tesla | CHURN |

| 7 | morgan | EXISTING |

| 8 | chase | CHURN |

| 9 | fitdata | NEW |

+----+---------+-----------+

9 rows in set (0.00 sec)

# Spotify Listening History

### Key Concepts

* Update aggregate table with event log.
* Temporary table & reusability.
* Update with join statement.
* Edge case: adding new user-song pair.
* Aggregation.

### Two Tables

You have a History table where you have date, user\_id, song\_id and count(tally). It shows at the end of each day how many times in her history a user has listened to a given song. So count is cumulative sum.

You have to update this on a daily basis based on a second Daily table that records in real time when a user listens to a given song.

Basically, at the end of each day, you go to this second table and pull a count of each user/song combination and then add this count to the first table that has the lifetime count.

### Sample Database

Load the database file [db.sql](https://github.com/NIteshx2/AdvancedSQL_Interview/blob/master/Interview/02_Spotify_Listening_History/db.sql) to localhost MySQL. A Spotify database will be created with two tables.

mysql < db.sql -uroot -p

mysql> SELECT \* from History;

+----+---------+---------+-------+

| id | user\_id | song\_id | tally |

+----+---------+---------+-------+

| 1 | shaw | rise | 2 |

| 2 | linda | lemon | 4 |

+----+---------+---------+-------+

2 rows in set (0.00 sec)

mysql> SELECT \* from Daily;

+----+---------+---------+---------------------+

| id | user\_id | song\_id | time\_stamp |

+----+---------+---------+---------------------+

| 1 | shaw | rise | 2019-03-01 05:33:08 |

| 2 | shaw | rise | 2019-03-01 16:00:00 |

| 3 | shaw | goodie | 2019-03-01 10:15:00 |

| 4 | linda | lemon | 2019-02-28 00:00:00 |

| 5 | mark | game | 2019-03-01 04:00:00 |

+----+---------+---------+---------------------+

5 rows in set (0.00 sec)

### Observation

* Note that the Daily table is a event-log. To update History, we need to aggregate the event log, grouping by user\_id and song\_id.
* A user may listen to a new song for the first time, in which case no existing (user\_id, song\_id) compound key pair exists in the History table. So we need an additional INSERT statement.

### Solution

Step 1. Build temporary table. For both the UPDATE and INSERT statements, we need the same aggregated information from the Daily table. So we can save it as a temporary table.

SET @now = "2019-03-01 00:00:00";

-- Create tamporary table

DROP TABLE IF EXISTS daily\_count;

CREATE TEMPORARY TABLE daily\_count

SELECT

user\_id

,song\_id

,COUNT(\*) AS tally

FROM Daily

WHERE DATEDIFF(@now, time\_stamp) = 0

GROUP BY user\_id, song\_id;

Check the temporary table.

mysql> SELECT \* FROM daily\_count;

+---------+---------+-------+

| user\_id | song\_id | tally |

+---------+---------+-------+

| mark | game | 1 |

| shaw | goodie | 1 |

| shaw | rise | 2 |

+---------+---------+-------+

3 rows in set (0.00 sec)

Step 2. Update existing pair. It's okay to join the temporary table with the History table during the update process, because History is independent of the temporary table.

UPDATE History AS uh

JOIN daily\_count AS dc

ON uh.user\_id = dc.user\_id

AND uh.song\_id = dc.song\_id

SET uh.tally = uh.tally + dc.tally;

Check if update is correct: shaw listened to rise twice on March 1. So the compound key is incremented by 2. On the other hand, linda did not listened to any song on March 1. So her number doesn't change.

mysql> SELECT \* FROM History;

+----+---------+---------+-------+

| id | user\_id | song\_id | tally |

+----+---------+---------+-------+

| 1 | shaw | rise | 4 |

| 2 | linda | lemon | 4 |

+----+---------+---------+-------+

2 rows in set (0.00 sec)

Step 3. Insert new pair. After updating existing (user\_id, song\_id) compound key pair, we need to insert new ones:

INSERT INTO History (user\_id, song\_id, tally)

SELECT

dc.user\_id

,dc.song\_id

,dc.tally

FROM daily\_count AS dc

LEFT JOIN History AS uh

ON dc.user\_id = uh.user\_id

AND dc.song\_id = uh.song\_id

WHERE uh.tally IS NULL;

Check that the insertions are correct.

mysql> SELECT \* FROM History;

+----+---------+---------+-------+

| id | user\_id | song\_id | tally |

+----+---------+---------+-------+

| 1 | shaw | rise | 4 |

| 2 | linda | lemon | 4 |

| 3 | mark | game | 1 |

| 4 | shaw | goodie | 1 |

+----+---------+---------+-------+

4 rows in set (0.00 sec)

# Monthly Active User

This post is inspired by this [page](https://www.programmerinterview.com/index.php/database-sql/practice-interview-question-2/). However, the solution from there is not completely correct, as they selected non-aggregated columns (Question 1). Also, DISTINCT keyword is not necessary in Question 2.

### Key Concepts

* Aggregate function
* Left join
* Removing duplicate
* Functional dependency

### Sample data

Load the database file [db.sql](https://github.com/NIteshx2/AdvancedSQL_Interview/blob/master/Interview/03_Monthly_Active_User/db.sql) to localhost MySQL. A MAU database will be created with two tables.

mysql < db.sql -uroot -p

mysql> SELECT \* FROM User;

+---------+---------+--------------+

| user\_id | name | phone\_num |

+---------+---------+--------------+

| jkog | Jing | 202-555-0176 |

| niceguy | Goodman | 202-555-0174 |

| sanhoo | Sanjay | 202-555-0100 |

| shaw123 | Shaw | 202-555-0111 |

+---------+---------+--------------+

4 rows in set (0.00 sec)

Every time a user logs in a new row is inserted into the UserHistory table with user\_id, current date and action (where action = "logged\_on").

mysql> SELECT \* FROM UserHistory;

+---------+------------+-----------+

| user\_id | date | action |

+---------+------------+-----------+

| shaw123 | 2019-02-20 | logged\_on |

| shaw123 | 2019-03-12 | signed\_up |

| sanhoo | 2019-02-27 | logged\_on |

| sanhoo | 2019-01-01 | logged\_on |

| niceguy | 2019-01-22 | logged\_on |

+---------+------------+-----------+

5 rows in set (0.00 sec)

### Q1: Find monthly active users.

Write a SQL query that returns the name, phone number and most recent date for any user that has logged in over the last 30 days (you can tell a user has logged in if the action field in UserHistory is set to "logged\_on").

SET @today := "2019-03-01";

SELECT

User.name

,User.phone\_num

,MAX(UserHistory.date)

FROM User, UserHistory

WHERE User.user\_id = UserHistory.user\_id

AND UserHistory.action = 'logged\_on'

AND UserHistory.date >= DATE\_SUB(@today, INTERVAL 30 DAY)

GROUP BY User.user\_id;

+--------+--------------+-----------------------+

| name | phone\_num | max(UserHistory.date) |

+--------+--------------+-----------------------+

| Sanjay | 202-555-0100 | 2019-02-27 |

| Shaw | 202-555-0111 | 2019-02-20 |

+--------+--------------+-----------------------+

2 rows in set (0.00 sec)

The above solution is only correct when phone\_num and name are functionally dependent on user\_id. That is, for every user\_id, there is a unique phone\_num and name, so we can get away with selecting non-aggregated columns (which are also not in group by clause).

Depending on database engine configuration, an error may be thrown when a selected column is neither aggregated nor in the group by clause. If we are certain that one-on-one mapping exists, we can add a aggregate function to the additional columns (MAX() or MIN()).

SET @today := "2019-03-01";

SELECT

MAX(u.name) -- functionally dependent on user\_id

,MAX(u.phone\_num) -- functionally dependent on user\_id

,MAX(h.date) AS recent\_date

FROM User AS u, UserHistory AS h

WHERE u.user\_id = h.user\_id

AND h.action = "logged\_on"

AND DATEDIFF(@today, h.date) <= 30 -- DATEDIFF(later, earlier)

GROUP BY u.user\_id

ORDER BY recent\_date;

+-------------+------------------+-------------+

| MAX(u.name) | MAX(u.phone\_num) | recent\_date |

+-------------+------------------+-------------+

| Shaw | 202-555-0111 | 2019-02-20 |

| Sanjay | 202-555-0100 | 2019-02-27 |

+-------------+------------------+-------------+

2 rows in set (0.01 sec)

Inner join also serves the purpose, and avoid making a cartesian product between two tables as in the cross join above (although query optimizer can take care of such trivial optimization, it's useful to know).

Why inner join: user\_id in UserHistory table is a foreign key referring to User table primary key. Meaning that it is a subset of the primary key column. There may exists users who never logged on, and never appeared in the UserHistory table. Since we are interested in monthly active users. It's safe to ignore those inactive users.

-- using inner join

SET @today := "2019-03-01";

SELECT

MAX(u.name) -- functionally dependent on user\_id

,MAX(u.phone\_num) -- functionally dependent on user\_id

,MAX(h.date) AS recent\_date

FROM User AS u

JOIN UserHistory AS h

ON u.user\_id = h.user\_id

WHERE h.action = "logged\_on"

AND DATEDIFF(@today, h.date) <= 30

GROUP BY u.user\_id

ORDER BY recent\_date;

If any selected column is not functionally dependent on the group by column, then unpredictable result may be returned, or error may be thrown. To avoid such trouble, only select aggregated columns and group by columns into a temporary tables, and join the temporary table with the original table to retrieve other desired columns.

### Q2. Find inactive users

Write a SQL query to determine which user\_ids in the User table are not contained in the UserHistory table (assume the UserHistory table has a subset of the user\_ids in User table). Do not use the SQL MINUS statement. Note: the UserHistory table can have multiple entries for each user\_id (Note that your SQL should be compatible with MySQL 5.0, and avoid using subqueries)

See [here](https://www.programmerinterview.com/index.php/database-sql/practice-interview-question-2-continued/) for a detailed walk-through. However, the solution is not totally correct. We don't need the DISTINCT keyword here. Because if a user\_id has no match in the UserHistory table, that row it returned only once.

SELECT \*

FROM User AS u

LEFT JOIN UserHistory AS h

ON u.user\_id = h.user\_id

WHERE h.user\_id IS NULL;

+---------+------+--------------+---------+------+--------+

| user\_id | name | phone\_num | user\_id | date | action |

+---------+------+--------------+---------+------+--------+

| jkog | Jing | 202-555-0176 | NULL | NULL | NULL |

+---------+------+--------------+---------+------+--------+

1 row in set (0.00 sec)

A less efficient approach is to retain valid users in a hashset. Remember that NOT IN requires full traversal of every element in the hashset for a single check (DISTINCT keyword turns the set into hash set, but makes no difference on the result).

SELECT \*

FROM User

WHERE user\_id NOT IN (

SELECT DISTINCT user\_id FROM UserHistory

);

+---------+------+--------------+

| user\_id | name | phone\_num |

+---------+------+--------------+

| jkog | Jing | 202-555-0176 |

+---------+------+--------------+

1 row in set (0.00 sec)

# Page Recommendation

This question is inspired by this quora [post](https://www.quora.com/How-can-I-prepare-for-a-case-analysis-interview-question-for-a-Facebook-data-scientist-position). Some very important key concepts are tested.

Write an SQL query that makes recommendations using the pages that your friends liked. Assume you have two tables: a two-column table of users and their friends, and a two-column table of users and the pages they liked. It should not recommend pages you already like.

### Key Concepts

* Accounding for mutual relationship (undirected edge) using relational table.
* Excluding pages users already follow.

### Sample data

Load the database file [db.sql](https://github.com/NIteshx2/AdvancedSQL_Interview/blob/master/Interview/04_Page_Recommendation/db.sql) to localhost MySQL. A Recommendation database will be created with two tables.

mysql < db.sql -uroot -p

mysql> SELECT \* FROM Friendship;

+----+---------+-----------+

| id | user\_id | friend\_id |

+----+---------+-----------+

| 1 | alice | bob |

| 2 | alice | charles |

| 3 | alice | david |

| 4 | bob | david |

+----+---------+-----------+

4 rows in set (0.00 sec)

mysql> SELECT \* FROM PageFollow;

+----+---------+----------+

| id | user\_id | page\_id |

+----+---------+----------+

| 1 | alice | google |

| 2 | bob | google |

| 3 | charles | google |

| 4 | bob | linkedin |

| 5 | charles | linkedin |

| 6 | david | linkedin |

| 7 | david | github |

| 8 | charles | github |

| 9 | alice | facebook |

| 10 | bob | facebook |

+----+---------+----------+

10 rows in set (0.00 sec)

### Solution

#### Step 1: Accounting Undirected Edge

Ask for clarification whether the Friendship table accounts for two directions. For example, if Alice is friend with Bob, are there two rows (Alice, Bob) and (Bob, Alice) in the table? If not (in this example), we need to union the table with itself.

This is necessary because when we aggregate over user\_id, we want to match to all friends that user\_id has. Alice will match to Bob, and Bob will match to Alice.

SELECT

user\_id

,friend\_id

FROM Friendship

UNION

SELECT

friend\_id

,user\_id

FROM Friendship;

+---------+-----------+

| user\_id | friend\_id |

+---------+-----------+

| alice | bob |

| alice | charles |

| alice | david |

| bob | david |

| bob | alice |

| charles | alice |

| david | alice |

| david | bob |

+---------+-----------+

8 rows in set (0.00 sec)

#### Step 2: Expand Pages

We are recommending pages based on what friends are following, so in this step, friend\_id is joined with PageFollow table.

WITH two\_way\_friendship AS (

SELECT

user\_id

,friend\_id

FROM Friendship

UNION

SELECT

friend\_id

,user\_id

FROM Friendship

)

SELECT

f.user\_id

,f.friend\_id

,p.page\_id

FROM two\_way\_friendship AS f

LEFT JOIN PageFollow AS p

ON f.friend\_id = p.user\_id

ORDER BY f.user\_id ASC, p.page\_id;

+---------+-----------+----------+

| user\_id | friend\_id | page\_id |

+---------+-----------+----------+

| alice | bob | facebook |

| alice | charles | github |

| alice | david | github |

| alice | charles | google |

| alice | bob | google |

| alice | bob | linkedin |

| alice | charles | linkedin |

| alice | david | linkedin |

| bob | alice | facebook |

| bob | david | github |

| bob | alice | google |

| bob | david | linkedin |

| charles | alice | facebook |

| charles | alice | google |

| david | alice | facebook |

| david | bob | facebook |

| david | alice | google |

| david | bob | google |

| david | bob | linkedin |

+---------+-----------+----------+

#### Step 3: Aggregation

We are recommending for each user, the pages with highest number of followers who are friends. In other word, we are counting friends for each (user\_id, page\_id). Be careful with what to put in GROUP BY and what to put in COUNT().

WITH two\_way\_friendship AS (

SELECT

user\_id

,friend\_id

FROM Friendship

UNION

SELECT

friend\_id

,user\_id

FROM Friendship

)

SELECT

f.user\_id

,p.page\_id

,COUNT(\*) AS friends\_follower

FROM two\_way\_friendship AS f

LEFT JOIN PageFollow AS p

ON f.friend\_id = p.user\_id

GROUP BY f.user\_id, p.page\_id

ORDER BY f.user\_id ASC, COUNT(\*) DESC;

+---------+----------+------------------+

| user\_id | page\_id | friends\_follower |

+---------+----------+------------------+

| alice | linkedin | 3 |

| alice | github | 2 |

| alice | google | 2 |

| alice | facebook | 1 |

| bob | google | 1 |

| bob | github | 1 |

| bob | facebook | 1 |

| bob | linkedin | 1 |

| charles | google | 1 |

| charles | facebook | 1 |

| david | google | 2 |

| david | facebook | 2 |

| david | linkedin | 1 |

+---------+----------+------------------+

13 rows in set (0.00 sec)

#### Step 4: De-duplicaton

We don't want to recommend pages user already likes. So we need to check for existance and exclude pages that are already liked.

In the final [solution](https://github.com/NIteshx2/AdvancedSQL_Interview/blob/master/Interview/04_Page_Recommendation/solution.sql) output, pages are ranked for each user by the number of friends who liked the page.

WITH two\_way\_friendship AS(

SELECT

user\_id

,friend\_id

FROM Friendship

UNION

SELECT

friend\_id

,user\_id

FROM Friendship

)

SELECT

f.user\_id

,p.page\_id

,COUNT(\*) AS friends\_follower

FROM two\_way\_friendship AS f

LEFT JOIN PageFollow AS p

ON f.friend\_id = p.user\_id

WHERE NOT EXISTS (

SELECT 1 FROM PageFollow AS p2

WHERE f.user\_id = p2.user\_id

AND p.page\_id = p2.page\_id

)

GROUP BY f.user\_id, p.page\_id

ORDER BY f.user\_id ASC, COUNT(\*) DESC;

+---------+----------+------------------+

| user\_id | page\_id | friends\_follower |

+---------+----------+------------------+

| alice | linkedin | 3 |

| alice | github | 2 |

| bob | github | 1 |

| charles | facebook | 1 |

| david | google | 2 |

| david | facebook | 2 |

+---------+----------+------------------+

6 rows in set (0.00 sec)

#### Optional: Tuple Predicate

More simply, we can use two-column pairs to check existance.

-- MySQL equivalent solution

WITH two\_way\_friendship AS(

SELECT

user\_id

,friend\_id

FROM Friendship

UNION

SELECT

friend\_id

,user\_id

FROM Friendship

)

SELECT

f.user\_id

,p.page\_id

,COUNT(\*) AS friends\_follower

FROM two\_way\_friendship AS f

LEFT JOIN PageFollow AS p

ON f.friend\_id = p.user\_id

WHERE (f.user\_id, p.page\_id) NOT IN (

SELECT user\_id, page\_id FROM PageFollow

)

GROUP BY f.user\_id, p.page\_id

ORDER BY f.user\_id ASC, COUNT(\*) DESC;

# Pivoting Numeric Data

We've seen a LeetCode problem on pivoting VARCHAR data [here](https://github.com/shawlu95/Beyond-LeetCode-SQL/tree/master/LeetCode/618_Students_Report_by_Geography). This notebook introduces the more common pivoting method: over NUMERIC data.

The trouble with pivoting VARCHAR data is that they cannot be aggregated over, and hence their position (row numebr) in the pivoted table must be determined by a RANK() function. For Numeric data, the process is much easier:

* Let's call the left most column of the pivoted table as the index column, and call the column that gets pivoted to multiple columns pivoted column.
* Note that the cardinality of the pivoted column determines how many more columns get added to the pivoted table. We must add those column one by one, either manually, or using dynamic query (more on that later).

The pivoting process is accomplished in two stages:

1. Adding columns: breaking the pivoted column into multiple columns. This can be accomplished using either CASE statement or self join. In this case, it is unrealistic to do self-join, because the number of rows is much greater than the cardinalities of the index and pivoted columns multiplied. See this [notebook](https://github.com/shawlu95/Beyond-LeetCode-SQL/tree/master/Interview/06_Pivoting_Text_Data) for an example of using self-join to pivot data.
2. Aggregation: reduce the number of rows. The number of rows will be equal to the cardinality of the index column.

### Data

In this example, we'll use real data on how I spent my money. The data are recorded using an iOS app, which conencts to a PHP server on my localhost, which interfaces with MySQL database. The data file contains one Expenses table, with a subset of the columns from the original table, covering one year of transaction data. You can see how I created the table [here](https://github.com/NIteshx2/AdvancedSQL_Interview/blob/master/Interview/05_Pivoting_Numeric_Data/Expenses.sql).

Load the database file [db.sql](https://github.com/NIteshx2/AdvancedSQL_Interview/blob/master/Interview/05_Pivoting_Numeric_Data/db.sql) to localhost MySQL. The Expenses table will be created in the Grocery database.

mysql < db.sql -uroot -p

Here is the schematic and a few rows of the table.

mysql> describe expenses;

+----------+---------------+------+-----+---------+-------+

| Field | Type | Null | Key | Default | Extra |

+----------+---------------+------+-----+---------+-------+

| category | varchar(14) | NO | | | |

| cost | decimal(12,2) | YES | | NULL | |

| time | datetime | NO | MUL | NULL | |

+----------+---------------+------+-----+---------+-------+

3 rows in set (0.01 sec)

mysql> SELECT \* FROM expenses LIMIT 5;

+------------+-------+---------------------+

| category | cost | time |

+------------+-------+---------------------+

| Social | 19.84 | 2018-01-02 13:30:27 |

| Book | 83.97 | 2018-01-03 13:30:27 |

| Food | 11.31 | 2018-01-03 13:30:27 |

| Stationary | 1.20 | 2018-01-03 13:30:27 |

| Food | 5.05 | 2018-01-03 13:30:27 |

+------------+-------+---------------------+

5 rows in set (0.00 sec)

## Objective

The unpivoted table contains 1083 rows (1083 records of transactions in year 2018). We want to pivot the table to have category as index column, and the time column as pivoted column, binned into 12 months. The cardinality of the category column is 25, The cardinality of the time column is 12 (12 months). The resulting table will have dimension 25 \* 12.

mysql> SELECT COUNT(DISTINCT category) FROM expenses;

+--------------------------+

| COUNT(DISTINCT category) |

+--------------------------+

| 25 |

+--------------------------+

1 row in set (0.00 sec)

#### Step 1. Adding Columns

Because there are too many rows, let's look at a few of them to get a sense of how it works. I'll take two days' data from November, and two days' data from December.

SELECT \* FROM Expenses

WHERE time BETWEEN "2018-12-01" AND "2018-12-02"

OR time BETWEEN "2018-11-01" AND "2018-11-03")

+---------------+-------+---------------------+

| category | cost | time |

+---------------+-------+---------------------+

| Insurance | 66.82 | 2018-11-02 12:30:27 |

| Automobile | 45.71 | 2018-11-02 12:30:27 |

| Food | 18.16 | 2018-12-01 13:30:27 |

| Communication | 6.98 | 2018-12-01 13:30:27 |

| Food | 1.66 | 2018-12-01 13:30:27 |

| Food | 5.92 | 2018-12-01 13:30:27 |

| Food | 6.24 | 2018-12-01 13:30:27 |

| Miscellany | 0.26 | 2018-12-01 13:30:27 |

+---------------+-------+---------------------+

8 rows in set (0.00 sec)

Using one CASE statement for each distinct value in the pivoted column ('Nov' and 'Dec' in this case). This creates two columns. For the 'Nov' column, we only want to retain data from November. For 'Dec' column, only retain data from December.

WITH tmp AS (

SELECT \* FROM Expenses

WHERE time BETWEEN "2018-12-01" AND "2018-12-02"

OR time BETWEEN "2018-11-01" AND "2018-11-03")

SELECT

category

,CASE WHEN EXTRACT(MONTH FROM time) = 12 THEN cost ELSE 0 END AS 'Dec'

,CASE WHEN EXTRACT(MONTH FROM time) = 11 THEN cost ELSE 0 END AS 'Nov'

FROM tmp

ORDER BY time;

+---------------+-------+-------+

| category | Dec | Nov |

+---------------+-------+-------+

| Insurance | 0 | 66.82 |

| Automobile | 0 | 45.71 |

| Food | 18.16 | 0 |

| Communication | 6.98 | 0 |

| Food | 1.66 | 0 |

| Food | 5.92 | 0 |

| Food | 6.24 | 0 |

| Miscellany | 0.26 | 0 |

+---------------+-------+-------+

8 rows in set (0.00 sec)

#### Step 2. Aggregation

Now, to calculate the total amount of money spent in each category, we simply aggregate over the category column!

WITH tmp AS (

SELECT \* FROM Expenses

WHERE time BETWEEN "2018-12-01" AND "2018-12-02"

OR time BETWEEN "2018-11-01" AND "2018-11-03")

SELECT

category

,SUM(CASE WHEN EXTRACT(MONTH FROM time) = 12 THEN cost ELSE 0 END) AS 'Dec'

,SUM(CASE WHEN EXTRACT(MONTH FROM time) = 11 THEN cost ELSE 0 END) AS 'Nov'

FROM tmp

GROUP BY category

ORDER BY category;

+---------------+-------+-------+

| category | Dec | Nov |

+---------------+-------+-------+

| Automobile | 0.00 | 45.71 |

| Communication | 6.98 | 0.00 |

| Food | 31.98 | 0.00 |

| Insurance | 0.00 | 66.82 |

| Miscellany | 0.26 | 0.00 |

+---------------+-------+-------+

5 rows in set (0.00 sec)

#### Pivot Full Table

Now we can apply the procedure to all 12 months! Sadly, as shown in the table below, I lost $14224.24 on the stock market when Nasdaq took a nose dive in October 2018.

SELECT

category

,SUM(CASE WHEN EXTRACT(MONTH FROM time) = 12 THEN cost ELSE 0 END) AS 'Dec'

,SUM(CASE WHEN EXTRACT(MONTH FROM time) = 11 THEN cost ELSE 0 END) AS 'Nov'

,SUM(CASE WHEN EXTRACT(MONTH FROM time) = 10 THEN cost ELSE 0 END) AS 'Oct'

,SUM(CASE WHEN EXTRACT(MONTH FROM time) = 09 THEN cost ELSE 0 END) AS 'Sep'

,SUM(CASE WHEN EXTRACT(MONTH FROM time) = 08 THEN cost ELSE 0 END) AS 'Aug'

,SUM(CASE WHEN EXTRACT(MONTH FROM time) = 07 THEN cost ELSE 0 END) AS 'Jul'

,SUM(CASE WHEN EXTRACT(MONTH FROM time) = 06 THEN cost ELSE 0 END) AS 'Jun'

,SUM(CASE WHEN EXTRACT(MONTH FROM time) = 05 THEN cost ELSE 0 END) AS 'May'

,SUM(CASE WHEN EXTRACT(MONTH FROM time) = 04 THEN cost ELSE 0 END) AS 'Apr'

,SUM(CASE WHEN EXTRACT(MONTH FROM time) = 03 THEN cost ELSE 0 END) AS 'Mar'

,SUM(CASE WHEN EXTRACT(MONTH FROM time) = 02 THEN cost ELSE 0 END) AS 'Feb'

,SUM(CASE WHEN EXTRACT(MONTH FROM time) = 01 THEN cost ELSE 0 END) AS 'Jan'

FROM Expenses

GROUP BY category

ORDER BY category;

+----------------+---------+---------+----------+---------+---------+---------+---------+---------+---------+---------+---------+---------+

| category | Dec | Nov | Oct | Sep | Aug | Jul | Jun | May | Apr | Mar | Feb | Jan |

+----------------+---------+---------+----------+---------+---------+---------+---------+---------+---------+---------+---------+---------+

| Accessory | 9.83 | 0.00 | 90.80 | 21.38 | 0.00 | 0.00 | 159.31 | 477.91 | 165.19 | 459.42 | 328.35 | 0.00 |

| Automobile | 59.57 | 94.20 | 347.99 | 1063.32 | 9877.37 | 58.88 | 0.00 | 3.11 | 0.00 | 0.00 | 0.00 | 209.50 |

| Book | 8.93 | 20.58 | 0.00 | 0.00 | 20.04 | 24.57 | 0.00 | 29.09 | 52.72 | 0.00 | 511.56 | 110.61 |

| Clothes | 0.00 | 202.03 | 0.00 | 0.00 | 131.19 | 10.36 | 726.79 | 213.45 | 671.34 | 83.73 | 58.76 | 67.16 |

| Communication | 23.62 | 16.64 | 16.64 | 15.18 | 5956.31 | 15.18 | 15.18 | 19.73 | 15.18 | 45.48 | 15.18 | 18.95 |

| Education | 4164.96 | 4164.96 | 4158.73 | 0.00 | 2222.32 | 588.82 | 3598.64 | 3598.64 | 3934.81 | 3881.69 | 3815.70 | 3815.70 |

| Electronics | 630.44 | 223.30 | 424.22 | 0.00 | 114.38 | 171.61 | 239.60 | 64.64 | 0.00 | 7.58 | 0.00 | 0.00 |

| Entertainment | 25.92 | 42.38 | 0.00 | 0.00 | 29.03 | 22.10 | 0.00 | 10.93 | 32.45 | 47.81 | 139.41 | 5.05 |

| Family | 104.50 | 22.85 | 126.04 | 0.00 | 0.00 | 1.50 | 0.00 | 0.00 | 0.00 | 64.77 | 20.78 | 63.98 |

| Food | 330.15 | 572.95 | 299.21 | 220.65 | 153.06 | 157.18 | 227.39 | 84.66 | 130.70 | 278.13 | 93.44 | 414.18 |

| Health | 44.87 | 35.91 | 40.60 | 71.11 | 586.95 | 159.73 | 87.45 | 64.08 | 81.52 | 64.12 | 307.01 | 139.65 |

| Home | 101.50 | 106.80 | 18.72 | 109.74 | 1656.58 | 296.79 | 813.37 | 88.71 | 0.00 | 154.45 | 35.39 | 18.12 |

| Housing | 0.00 | 1264.03 | 1295.14 | 1247.27 | 2571.53 | 2573.26 | 1037.91 | 1037.91 | 1061.77 | 1233.07 | 1233.07 | 1372.74 |

| Insurance | 150.81 | 150.81 | 150.81 | 150.81 | 251.01 | 0.00 | 526.91 | 526.91 | 526.91 | 558.69 | 558.69 | 771.24 |

| Invest | 587.29 | 2555.41 | 14224.24 | 731.55 | 768.84 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 |

| Legal | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 6.24 | 26.92 | 0.00 | 505.45 | 1.81 | 0.00 | 0.00 |

| Miscellany | 6.48 | 76.25 | 26.61 | 0.00 | 0.26 | 0.00 | 762.15 | 138.07 | 0.00 | 10.12 | 0.00 | 47.61 |

| Pen | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 75.47 | 105.11 | 0.00 | 0.00 | 0.00 | 0.00 |

| Sartorial | 0.00 | 72.09 | 0.00 | 0.00 | 4.54 | 0.00 | 81.15 | 125.37 | 88.36 | 135.86 | 21.53 | 0.00 |

| Shoes | 497.89 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 580.91 | 0.00 | 81.09 | 0.00 | 0.00 | 12.79 |

| Social | 86.14 | 265.27 | 120.58 | 167.03 | 165.50 | 49.41 | 286.32 | 109.33 | 228.65 | 185.25 | 71.38 | 279.38 |

| Software | 117.37 | 40.73 | 66.60 | 40.73 | 41.83 | 318.07 | 51.68 | 46.48 | 46.48 | 39.24 | 20.24 | 90.08 |

| Stationary | 0.00 | 0.00 | 14.57 | 4.05 | 25.99 | 0.00 | 84.61 | 17.56 | 20.77 | 29.64 | 3.33 | 13.52 |

| Tax | 49.49 | 0.00 | 4.72 | 0.46 | 102.29 | 51.91 | 122.77 | 20.19 | 71.93 | 19.17 | 8.15 | 6.07 |

| Transportation | 1694.83 | 0.00 | 14.55 | 0.00 | 81.97 | 248.49 | 86.60 | 89.61 | 45.31 | 303.49 | 14.32 | 148.53 |

+----------------+---------+---------+----------+---------+---------+---------+---------+---------+---------+---------+---------+---------+

25 rows in set (0.00 sec)

# Pivoting Text Data

Different from the Expenses table from the earlier [note](https://github.com/shawlu95/Beyond-LeetCode-SQL/tree/master/Interview/05_Pivoting_Numeric_Data), this table contains text data, and cannot be summed or averaged over. A different approach is required to reduce number of tables. Still, just as in pivoting numeric data, we need to go through two steps.

1. Adding columns using self-join or switch statement. We'll study both approaches in this notebook.
2. Reduce number of rows to the cardinality of index column.

Load the database file [db.sql](https://github.com/NIteshx2/AdvancedSQL_Interview/blob/master/Interview/06_Pivoting_Text_Data/db.sql) to localhost MySQL. The CourseGrade table will be created in the Practice database. The table CourseGrade contains letter grade of five students on three courses. It has exactly |pivot column| \* |index column| = 3 \* 5 = 15 rows, and the info column, grade cannot be summed as we did before with numeric data.

mysql < db.sql -uroot -p

Different from the [LeetCode](https://github.com/shawlu95/Beyond-LeetCode-SQL/tree/master/LeetCode/618_Students_Report_by_Geography) problem, here we do have an index column name which determines the row number for the info column grade. So we do not have to reindex the table using ROW\_NUMBER().

mysql> select \* from coursegrade;

+---------+--------+-------+

| name | course | grade |

+---------+--------+-------+

| Alice | CS106B | A |

| Alice | CS229 | A |

| Alice | CS224N | B |

| Bob | CS106B | C |

| Bob | CS229 | F |

| Bob | CS224N | F |

| Charlie | CS106B | B |

| Charlie | CS229 | B |

| Charlie | CS224N | A |

| David | CS106B | C |

| David | CS229 | C |

| David | CS224N | D |

| Elsa | CS106B | B |

| Elsa | CS229 | B |

| Elsa | CS224N | A |

+---------+--------+-------+

15 rows in set (0.00 sec)

### Pivot Using Self-join

Let's look at a subset of the data.

SELECT \* FROM CourseGrade WHERE name IN ('Alice', 'Bob')

+-------+--------+-------+

| name | course | grade |

+-------+--------+-------+

| Alice | CS106B | A |

| Alice | CS229 | A |

| Alice | CS224N | B |

| Bob | CS106B | C |

| Bob | CS229 | F |

| Bob | CS224N | F |

+-------+--------+-------+

6 rows in set (0.00 sec)

Unconditional cross join produces 6 \* 6 = 36 rows. Self-joining the table using the name results in 3 courses \* 3 courses = 9 rows for each student. For each student, every class is matched to three other classes.

WITH tmp AS (

SELECT \* FROM CourseGrade WHERE name IN ('Alice', 'Bob')

)

SELECT

\*

FROM tmp AS t1, tmp AS t2

WHERE t1.name = t2.name;

+-------+--------+-------+-------+--------+-------+

| name | course | grade | name | course | grade |

+-------+--------+-------+-------+--------+-------+

| Alice | CS106B | A | Alice | CS106B | A |

| Alice | CS106B | A | Alice | CS229 | A |

| Alice | CS106B | A | Alice | CS224N | B |

| Alice | CS229 | A | Alice | CS106B | A |

| Alice | CS229 | A | Alice | CS229 | A |

| Alice | CS229 | A | Alice | CS224N | B |

| Alice | CS224N | B | Alice | CS106B | A |

| Alice | CS224N | B | Alice | CS229 | A |

| Alice | CS224N | B | Alice | CS224N | B |

| Bob | CS106B | C | Bob | CS106B | C |

| Bob | CS106B | C | Bob | CS229 | F |

| Bob | CS106B | C | Bob | CS224N | F |

| Bob | CS229 | F | Bob | CS106B | C |

| Bob | CS229 | F | Bob | CS229 | F |

| Bob | CS229 | F | Bob | CS224N | F |

| Bob | CS224N | F | Bob | CS106B | C |

| Bob | CS224N | F | Bob | CS229 | F |

| Bob | CS224N | F | Bob | CS224N | F |

+-------+--------+-------+-------+--------+-------+

18 rows in set (0.00 sec)

\*/

For each student, we want one column for each course, so we can directly filter on the course name. Filtering one table's course name reduces the number of rows by a factor of 3. Filtering both tables reduces by a factor of 3 \* 3.

WITH tmp AS (

SELECT \* FROM CourseGrade WHERE name IN ('Alice', 'Bob')

)

SELECT

\*

FROM tmp AS t1, tmp AS t2

WHERE t1.name = t2.name

AND t1.course = 'CS106B';

+-------+--------+-------+-------+--------+-------+

| name | course | grade | name | course | grade |

+-------+--------+-------+-------+--------+-------+

| Alice | CS106B | A | Alice | CS106B | A |

| Alice | CS106B | A | Alice | CS229 | A |

| Alice | CS106B | A | Alice | CS224N | B |

| Bob | CS106B | C | Bob | CS106B | C |

| Bob | CS106B | C | Bob | CS229 | F |

| Bob | CS106B | C | Bob | CS224N | F |

+-------+--------+-------+-------+--------+-------+

6 rows in set (0.00 sec)

WITH tmp AS (

SELECT \* FROM CourseGrade WHERE name IN ('Alice', 'Bob')

)

SELECT

\*

FROM tmp AS t1, tmp AS t2

WHERE t1.name = t2.name

AND t1.course = 'CS106B'

AND t2.course = 'CS229'

;

+-------+--------+-------+-------+--------+-------+

| name | course | grade | name | course | grade |

+-------+--------+-------+-------+--------+-------+

| Alice | CS106B | A | Alice | CS229 | A |

| Bob | CS106B | C | Bob | CS229 | F |

+-------+--------+-------+-------+--------+-------+

2 rows in set (0.00 sec)

We can easily generalize to three tables, and possibly more.

WITH tmp AS (

SELECT \* FROM CourseGrade WHERE name IN ('Alice', 'Bob')

)

SELECT

\*

FROM tmp AS t1, tmp AS t2, tmp AS t3

WHERE t1.name = t2.name

AND t2.name = t3.name

AND t1.course = 'CS106B'

AND t2.course = 'CS229'

AND t3.course = 'CS224N';

+-------+--------+-------+-------+--------+-------+-------+--------+-------+

| name | course | grade | name | course | grade | name | course | grade |

+-------+--------+-------+-------+--------+-------+-------+--------+-------+

| Alice | CS106B | A | Alice | CS229 | A | Alice | CS224N | B |

| Bob | CS106B | C | Bob | CS229 | F | Bob | CS224N | F |

+-------+--------+-------+-------+--------+-------+-------+--------+-------+

2 rows in set (0.00 sec)

Notice that by setting the filtering criteria, we are setting the value of the additional columns! To make it clearer, simply rename the column title with the course title, and get rid of the course column, since it's redundant.

WITH tmp AS (

SELECT \* FROM CourseGrade WHERE name IN ('Alice', 'Bob')

)

SELECT

t1.name

,t1.grade AS 'CS106B'

,t2.grade AS 'CS229'

,t3.grade AS 'CS224N'

FROM tmp AS t1, tmp AS t2, tmp AS t3

WHERE t1.name = t2.name

AND t2.name = t3.name

AND t1.course = 'CS106B'

AND t2.course = 'CS229'

AND t3.course = 'CS224N';

+-------+--------+-------+--------+

| name | CS106B | CS229 | CS224N |

+-------+--------+-------+--------+

| Alice | A | A | B |

| Bob | C | F | F |

+-------+--------+-------+--------+

2 rows in set (0.00 sec)

Now we can remove the temporary table and pivot the entire table.

SELECT

t1.name

,t1.grade AS 'CS106B'

,t2.grade AS 'CS229'

,t3.grade AS 'CS224N'

FROM CourseGrade AS t1, CourseGrade AS t2, CourseGrade AS t3

WHERE t1.name = t2.name

AND t2.name = t3.name

AND t1.course = 'CS106B'

AND t2.course = 'CS229'

AND t3.course = 'CS224N';

+---------+--------+-------+--------+

| name | CS106B | CS229 | CS224N |

+---------+--------+-------+--------+

| Alice | A | A | B |

| Bob | C | F | F |

| Charlie | B | B | A |

| David | C | C | D |

| Elsa | B | B | A |

+---------+--------+-------+--------+

5 rows in set (0.00 sec)

To boost efficiency, we may replace cross join with inner join, and pre-filter on the course title before join.

SELECT

t1.name

,t1.grade AS 'CS106B'

,t2.grade AS 'CS229'

,t3.grade AS 'CS224N'

FROM CourseGrade AS t1

JOIN CourseGrade AS t2

ON t1.course = 'CS106B'

AND t2.course = 'CS229'

AND t1.name = t2.name

JOIN CourseGrade AS t3

ON t3.course = 'CS224N'

AND t2.name = t3.name;

+---------+--------+-------+--------+

| name | CS106B | CS229 | CS224N |

+---------+--------+-------+--------+

| Alice | A | A | B |

| Bob | C | F | F |

| Charlie | B | B | A |

| David | C | C | D |

| Elsa | B | B | A |

+---------+--------+-------+--------+

5 rows in set (0.00 sec)

### Bonus: Using Case Statement

We can use the case statement, as described in the previous [note](https://github.com/NIteshx2/AdvancedSQL_Interview/blob/master/Interview/05_Pivoting_Numeric_Data). First, we add columns, using one case statement for each column.

SELECT

\*

,CASE WHEN course = 'CS106B' THEN grade ELSE NULL END AS 'CS106B'

,CASE WHEN course = 'CS229' THEN grade ELSE NULL END AS 'CS229'

,CASE WHEN course = 'CS224N' THEN grade ELSE NULL END AS 'CS224N'

FROM CourseGrade;

+---------+--------+-------+--------+-------+--------+

| name | course | grade | CS106B | CS229 | CS224N |

+---------+--------+-------+--------+-------+--------+

| Alice | CS106B | A | A | NULL | NULL |

| Alice | CS229 | A | NULL | A | NULL |

| Alice | CS224N | B | NULL | NULL | B |

| Bob | CS106B | C | C | NULL | NULL |

| Bob | CS229 | F | NULL | F | NULL |

| Bob | CS224N | F | NULL | NULL | F |

| Charlie | CS106B | B | B | NULL | NULL |

| Charlie | CS229 | B | NULL | B | NULL |

| Charlie | CS224N | A | NULL | NULL | A |

| David | CS106B | C | C | NULL | NULL |

| David | CS229 | C | NULL | C | NULL |

| David | CS224N | D | NULL | NULL | D |

| Elsa | CS106B | B | B | NULL | NULL |

| Elsa | CS229 | B | NULL | B | NULL |

| Elsa | CS224N | A | NULL | NULL | A |

+---------+--------+-------+--------+-------+--------+

15 rows in set (0.00 sec)

Next, we reduce the numebr of rows to the cardinality of index column, using aggregate function. The redundant column course is dropped out. Since each group only contains one valid cell that is not NULL, using either MAX() or MIN() gives the same result. Just don't use AVG(), SUM() or COUNT().

SELECT

name

,MAX(CASE WHEN course = 'CS106B' THEN grade ELSE NULL END) AS 'CS106B'

,MAX(CASE WHEN course = 'CS229' THEN grade ELSE NULL END) AS 'CS229'

,MAX(CASE WHEN course = 'CS224N' THEN grade ELSE NULL END) AS 'CS224N'

FROM CourseGrade

GROUP BY name;

+---------+--------+-------+--------+

| name | CS106B | CS229 | CS224N |

+---------+--------+-------+--------+

| Alice | A | A | B |

| Bob | C | F | F |

| Charlie | B | B | A |

| David | C | C | D |

| Elsa | B | B | A |

+---------+--------+-------+--------+

5 rows in set (0.00 sec)

SELECT

name

,MIN(CASE WHEN course = 'CS106B' THEN grade ELSE NULL END) AS 'CS106B'

,MIN(CASE WHEN course = 'CS229' THEN grade ELSE NULL END) AS 'CS229'

,MIN(CASE WHEN course = 'CS224N' THEN grade ELSE NULL END) AS 'CS224N'

FROM CourseGrade

GROUP BY name;

+---------+--------+-------+--------+

| name | CS106B | CS229 | CS224N |

+---------+--------+-------+--------+

| Alice | A | A | B |

| Bob | C | F | F |

| Charlie | B | B | A |

| David | C | C | D |

| Elsa | B | B | A |

+---------+--------+-------+--------+

5 rows in set (0.00 sec)

**Unpivoting Table**

This is the reverse process of pivoting. In effect, we are reducing the number of columns, and increasing the number of rows. The key is to define an auxiliary table. In this notebook, we will un-pivot both the numeric pivot [table](https://github.com/shawlu95/Beyond-LeetCode-SQL/tree/master/Interview/05_Pivoting_Numeric_Data) and the text pivot [table](https://github.com/shawlu95/Beyond-LeetCode-SQL/tree/master/Interview/06_Pivoting_Text_Data).

An auxiliary table is a temporary table that contains a single columns containing the name of the columns to be dropped from the pivot table.

The reversal is accomplished by cross joining with an auxiliary table, to inrease the number of rows. Then we apply a case statement on the auxiliary table's value, and select the to-be-dropped columns correspondingly.

In this notebook, we'll go over two examples: unpivoting the numeric table and text table we accomplished in the earlier notebooks. You'll see that, by aggregating over numeric data, we have permanently lost information, and cannot fully un-pivot back to the original state.

Before getting started, load the pivot table by running the following [script](https://github.com/NIteshx2/AdvancedSQL_Interview/blob/master/Interview/07_Unpivoting_Tables/pivot_table.sql).

mysql < pivot\_table.sql -uroot -p

**Unpivoting Text Data**

**Step 1. Build the auxiliary Table**

Switch to Practice database:

USE Practice;

Different SQL server gives you different syntax in building auxiliary table. In MySQL, the process is rather verbose.

SELECT \* FROM (

SELECT 'CS106B' AS course\_name

UNION ALL

SELECT 'CS229'

UNION ALL

SELECT 'CS224N'

) aux;

+-------------+

| course\_name |

+-------------+

| CS106B |

| CS229 |

| CS224N |

+-------------+

3 rows in set (0.00 sec)

**Step 2. Cross Join with Pivot Table**

Notice that this will uncover the numebr of rows to 15 (5 students \* 3 courses), which is where we started with.

SELECT \* FROM

course\_grade\_pivoted,

(

SELECT 'CS106B' AS course

UNION ALL

SELECT 'CS229'

UNION ALL

SELECT 'CS224N'

) aux;

+---------+--------+-------+--------+-------------+

| name | CS106B | CS229 | CS224N | course |

+---------+--------+-------+--------+-------------+

| Alice | A | A | B | CS106B |

| Alice | A | A | B | CS229 |

| Alice | A | A | B | CS224N |

| Bob | C | F | F | CS106B |

| Bob | C | F | F | CS229 |

| Bob | C | F | F | CS224N |

| Charlie | B | B | A | CS106B |

| Charlie | B | B | A | CS229 |

| Charlie | B | B | A | CS224N |

| David | C | C | D | CS106B |

| David | C | C | D | CS229 |

| David | C | C | D | CS224N |

| Elsa | B | B | A | CS106B |

| Elsa | B | B | A | CS229 |

| Elsa | B | B | A | CS224N |

+---------+--------+-------+--------+-------------+

15 rows in set (0.00 sec)

We're simply multiplying each row in the pivot table three times, with each of the three courses. This step results in redundant data. For each student, the CS206B, CS229, CS224N columns are all identical. But notice that we have recovered back to 15 rows, and the original course column!

SELECT \* FROM CourseGrade;

+---------+--------+-------+

| name | course | grade |

+---------+--------+-------+

| Alice | CS106B | A |

| Alice | CS229 | A |

| Alice | CS224N | B |

| Bob | CS106B | C |

| Bob | CS229 | F |

| Bob | CS224N | F |

| Charlie | CS106B | B |

| Charlie | CS229 | B |

| Charlie | CS224N | A |

| David | CS106B | C |

| David | CS229 | C |

| David | CS224N | D |

| Elsa | CS106B | B |

| Elsa | CS229 | B |

| Elsa | CS224N | A |

+---------+--------+-------+

15 rows in set (0.00 sec)

The only thing we need to do know is to combine the three columns to one, extracting where the *column title* matches the row value of the *course* column. For each student, this is precisely the diagonal line of the square mateix!

| **name** | **CS106B** | **CS229** | **CS224N** | **course** |
| --- | --- | --- | --- | --- |
| Alice | **A** | A | B | CS106B |
| Alice | A | **A** | B | CS229 |
| Alice | A | A | **B** | CS224N |

We can use a case statement to condition on the value of the course column, and extract the corresponding value from the matching columns.

SELECT

name

,aux.course

,CASE aux.course

WHEN 'CS106B' THEN CS106B

WHEN 'CS229' THEN CS229

WHEN 'CS224N' THEN CS224N

END AS grade

FROM course\_grade\_pivoted,

(

SELECT 'CS106B' AS course

UNION ALL

SELECT 'CS229'

UNION ALL

SELECT 'CS224N'

) aux;

+---------+--------+-------+

| name | course | grade |

+---------+--------+-------+

| Alice | CS106B | A |

| Alice | CS229 | A |

| Alice | CS224N | B |

| Bob | CS106B | C |

| Bob | CS229 | F |

| Bob | CS224N | F |

| Charlie | CS106B | B |

| Charlie | CS229 | B |

| Charlie | CS224N | A |

| David | CS106B | C |

| David | CS229 | C |

| David | CS224N | D |

| Elsa | CS106B | B |

| Elsa | CS229 | B |

| Elsa | CS224N | A |

+---------+--------+-------+

15 rows in set (0.00 sec)

**Unpivoting Numeric Data**

To unpivote the table from this [notebook](https://github.com/NIteshx2/AdvancedSQL_Interview/blob/master/Interview/05_Pivoting_Numeric_Data), we apply the same two-step technique. However, because the pivot table was constructed by summing over groups, we are unable to uncover the 1083 rows. Instead, we can only recover 25 categories \* 12 months = 300 rows.

Switch to Grocery database:

USE Grocery;

Here we show 24 rows, for 'Social' and 'Book' categories.

SELECT

category

,aux.month

,CASE aux.month

WHEN 'Jan' THEN Jan

WHEN 'Feb' THEN Feb

WHEN 'Mar' THEN Mar

WHEN 'Apr' THEN Apr

WHEN 'May' THEN May

WHEN 'Jun' THEN Jun

WHEN 'Jul' THEN Jul

WHEN 'Aug' THEN Aug

WHEN 'Sep' THEN Sep

WHEN 'Oct' THEN Oct

WHEN 'Nov' THEN Nov

WHEN 'Dec' THEN Dec\_

END AS month

FROM expenses\_pivoted,

(

SELECT 'Jan' AS month

UNION ALL

SELECT 'Feb'

UNION ALL

SELECT 'Mar'

UNION ALL

SELECT 'Apr'

UNION ALL

SELECT 'May'

UNION ALL

SELECT 'Jun'

UNION ALL

SELECT 'Jul'

UNION ALL

SELECT 'Aug'

UNION ALL

SELECT 'Sep'

UNION ALL

SELECT 'Oct'

UNION ALL

SELECT 'Nov'

UNION ALL

SELECT 'Dec'

) AS aux

LIMIT 24;

+----------+-------+--------+

| category | month | month |

+----------+-------+--------+

| Social | Jan | 279.38 |

| Social | Feb | 71.38 |

| Social | Mar | 185.25 |

| Social | Apr | 228.65 |

| Social | May | 109.33 |

| Social | Jun | 286.32 |

| Social | Jul | 49.41 |

| Social | Aug | 165.50 |

| Social | Sep | 167.03 |

| Social | Oct | 120.58 |

| Social | Nov | 265.27 |

| Social | Dec | 86.14 |

| Book | Jan | 110.61 |

| Book | Feb | 511.56 |

| Book | Mar | 0.00 |

| Book | Apr | 52.72 |

| Book | May | 29.09 |

| Book | Jun | 0.00 |

| Book | Jul | 24.57 |

| Book | Aug | 20.04 |

| Book | Sep | 0.00 |

| Book | Oct | 0.00 |

| Book | Nov | 20.58 |

| Book | Dec | 8.93 |

+----------+-------+--------+

24 rows in set (0.00 sec)

In case you're wondering why I write 'Dec\_' instead of 'Dec'. It turns out that 'Dec' is a reserved keyword. It is a function that converts a binary string argument into a numeric value.

# Group by Bins

One common task is to put rows into bins, meaning that there does not exist a conveninet column for us to GROUP BY. In this notebook, we will use the [classicmodels database](https://github.com/NIteshx2/AdvancedSQL_Interview/blob/master/databases/classicmodels/mysqlsampledatabase.sql). Load the database:

mysql < mysqlsampledatabase.sql -uroot -p

It contains a customers table. We will put customers into different class of credit limit.

* 0~50,000
* 50,001~100,000
* over 100,000

### Step 1. Create Column to Group By

SELECT

customerName

,CASE

WHEN creditLimit BETWEEN 0 AND 50000 THEN '0 ~ 50k'

WHEN creditLimit BETWEEN 50001 AND 100000 THEN '50 ~ 100k'

ELSE '> 100k'

END AS credit\_range

FROM customers

LIMIT 5;

+----------------------------+--------------+

| customerName | credit\_range |

+----------------------------+--------------+

| Atelier graphique | 0 ~ 50k |

| Signal Gift Stores | 50 ~ 100k |

| Australian Collectors, Co. | > 100k |

| La Rochelle Gifts | > 100k |

| Baane Mini Imports | 50 ~ 100k |

+----------------------------+--------------+

### Step 2. Group by Bin

SELECT

CASE

WHEN creditLimit BETWEEN 0 AND 50000 THEN '0 ~ 50k'

WHEN creditLimit BETWEEN 50001 AND 100000 THEN '50 ~ 100k'

ELSE '> 100k'

END AS credit\_range

,COUNT(\*) AS customer\_tally

FROM customers

GROUP BY credit\_range;

+--------------+----------------+

| credit\_range | customer\_tally |

+--------------+----------------+

| 0 ~ 50k | 37 |

| 50 ~ 100k | 60 |

| > 100k | 25 |

+--------------+----------------+

3 rows in set (0.00 sec)

### Parting thought

This type of question can easily generalize to a broad range of topics. For example, what is the distribution of user's age, what is the distribution of revenue from each online order?

# Consecutive Active Users

Given timestamps of logins, figure out how many people on Facebook were active ALL seven days of a week on a mobile phone.

Different from the monthly active user [problem](https://github.com/shawlu95/Beyond-LeetCode-SQL/tree/master/Interview/03_Monthly_Active_User), in which active user is defined by those who logged on **at least once** in the past month, in this problem we define acive users as those who logged on in the past **7 consecutive days**. In this problem, we'll require only 3 days. The logic is the same.

### Key Concepts

* Self join.
* Window functions.

### Table

mysql> describe Login;

+---------+---------+------+-----+---------+----------------+

| Field | Type | Null | Key | Default | Extra |

+---------+---------+------+-----+---------+----------------+

| id | int(11) | NO | PRI | NULL | auto\_increment |

| user\_id | int(11) | YES | | NULL | |

| ts | date | YES | | NULL | |

+---------+---------+------+-----+---------+----------------+

3 rows in set (0.00 sec)

mysql> select \* from Login;

+----+---------+------------+

| id | user\_id | ts |

+----+---------+------------+

| 1 | 1 | 2019-02-14 |

| 2 | 1 | 2019-02-13 |

| 3 | 1 | 2019-02-12 |

| 4 | 1 | 2019-02-11 |

| 5 | 2 | 2019-02-14 |

| 6 | 2 | 2019-02-12 |

| 7 | 2 | 2019-02-11 |

| 8 | 2 | 2019-02-10 |

| 9 | 3 | 2019-02-14 |

| 10 | 3 | 2019-02-12 |

| 11 | 4 | 2019-02-09 |

| 12 | 4 | 2019-02-08 |

| 13 | 4 | 2019-02-08 |

| 14 | 4 | 2019-02-07 |

+----+---------+------------+

14 rows in set (0.00 sec)

### Self-join Method

Directly joining tables against itself 7 times causes row number to explode. Since we are not differentiating same user's multiple logins in a day, we only need to keep one record for each user per day before joining the table. Because the ts column is date type, we only need a DISTINCT keyword to filter the duplicates. If the column is datetime type, we would need to transform it into date type first.

WITH tmp AS (

SELECT DISTINCT user\_id, ts FROM Login)

SELECT

\*

FROM tmp AS d0

JOIN tmp AS d1

ON d0.user\_id = d1.user\_id

AND DATEDIFF(d0.ts, d1.ts) = 1

JOIN tmp AS d2

ON d2.user\_id = d1.user\_id

AND DATEDIFF(d0.ts, d2.ts) = 2;

-- AND DATEDIFF(d1.ts, d2.ts) = 1 -- only need one of the condition, because we are using inner join (no NULL)

+---------+------------+---------+------------+---------+------------+

| user\_id | ts | user\_id | ts | user\_id | ts |

+---------+------------+---------+------------+---------+------------+

| 1 | 2019-02-14 | 1 | 2019-02-13 | 1 | 2019-02-12 |

| 1 | 2019-02-13 | 1 | 2019-02-12 | 1 | 2019-02-11 |

| 2 | 2019-02-12 | 2 | 2019-02-11 | 2 | 2019-02-10 |

| 4 | 2019-02-09 | 4 | 2019-02-08 | 4 | 2019-02-07 |

+---------+------------+---------+------------+---------+------------+

4 rows in set (0.00 sec)

Did you see any problem? The query returns all users who have **ever** been active for 3 days in **all** history. The question asks for users who have been active in the **most recent** 3 days only. So we need to add condition on the d0.ts column. Preferably, we use pre-filter instead of post-filter. Only one user satisfies out conditions.

@now = '2019-02-04';

WITH tmp AS (

SELECT DISTINCT user\_id, ts FROM Login)

SELECT \*

FROM tmp AS d0

JOIN tmp AS d1

ON d0.ts = @now

AND d0.user\_id = d1.user\_id

AND DATEDIFF(d0.ts, d1.ts) = 1

JOIN tmp AS d2

ON d2.user\_id = d1.user\_id

AND DATEDIFF(d0.ts, d2.ts) = 2;

+---------+------------+---------+------------+---------+------------+

| user\_id | ts | user\_id | ts | user\_id | ts |

+---------+------------+---------+------------+---------+------------+

| 1 | 2019-02-14 | 1 | 2019-02-13 | 1 | 2019-02-12 |

+---------+------------+---------+------------+---------+------------+

1 row in set (0.00 sec)

Equivalently, we can pre-filter the source table into several sub-tables before joining them. This solution also works for datetime data type.

SET @now = "2019-02-14";

WITH tmp AS (

SELECT DISTINCT user\_id, ts FROM Login)

SELECT \*

FROM tmp AS d0

JOIN tmp AS d1

ON d0.ts = @now

AND DATEDIFF(@now, d1.ts) = 1

AND d0.user\_id = d1.user\_id

JOIN tmp AS d2

ON DATEDIFF(@now, d2.ts) = 2

AND d2.user\_id = d1.user\_id;

+----+---------+------------+----+---------+------------+----+---------+------------+

| id | user\_id | ts | id | user\_id | ts | id | user\_id | ts |

+----+---------+------------+----+---------+------------+----+---------+------------+

| 1 | 1 | 2019-02-14 | 2 | 1 | 2019-02-13 | 3 | 1 | 2019-02-12 |

+----+---------+------------+----+---------+------------+----+---------+------------+

1 row in set (0.00 sec)

Note that by using inner join, we are incrementally filtering out users who are not active in a day. This is okay because we only need user\_id. If we were to calculate the proportion of active user, we need to do **full join** (left join is not sufficient). The total number of rows returned by the full join is the number of users who have been active in at least one of the past 3 days.

### Window Method

The window function takes a different mental twist. Instead of finding customers who are active on day 0, day 1, ... day N. We are asking the following question:

* Q1: How many days ago was the last log in (earlier than today)?
* Q2: How many days ago was the second last log in (earlier than the last log in)?
* ...

If the answer to Q1 is 1, answer to Q2 is 2, then the customer is active in the most recent 3 days.

Note that for this logic to work, we must have **one** record per day for each customers. If we have multiple records per day, LAG function will return 0 for those duplicates.

SET @now = "2019-02-14";

SELECT

user\_id

,DATEDIFF(@now, LAG(ts, 1) OVER w) AS day\_from\_pre1

,DATEDIFF(@now, LAG(ts, 2) OVER w) AS day\_from\_pre2

FROM Login

WINDOW w AS (PARTITION BY user\_id ORDER BY ts);

+---------+---------------+---------------+

| user\_id | day\_from\_pre1 | day\_from\_pre2 |

+---------+---------------+---------------+

| 1 | NULL | NULL |

| 1 | 3 | NULL |

| 1 | 2 | 3 |

| 1 | 1 | 2 |

| 2 | NULL | NULL |

| 2 | 4 | NULL |

| 2 | 3 | 4 |

| 2 | 2 | 3 |

| 3 | NULL | NULL |

| 3 | 2 | NULL |

| 4 | NULL | NULL |

| 4 | 7 | NULL |

| 4 | 6 | 7 |

| 4 | 6 | 6 |

+---------+---------------+---------------+

14 rows in set (0.00 sec)

**Warning**: because WHERE is evaluated before window function, the following table returns returns nothing in the window functions.

Window functions are permitted only in the SELECT list and the ORDER BY clause of the query. They are forbidden elsewhere, such as in GROUP BY, HAVING and WHERE clauses. This is because they logically execute after the processing of those clauses. Also, window functions execute after regular aggregate functions.

SET @now = "2019-02-14";

SELECT

user\_id

,ts

,DATEDIFF(@now, LAG(ts, 1) OVER (PARTITION BY user\_id ORDER BY ts)) AS day\_from\_pre1

,DATEDIFF(@now, LAG(ts, 2) OVER (PARTITION BY user\_id ORDER BY ts)) AS day\_from\_pre2

FROM Login

WHERE ts = @now;

+---------+------------+---------------+---------------+

| user\_id | ts | day\_from\_pre1 | day\_from\_pre2 |

+---------+------------+---------------+---------------+

| 1 | 2019-02-14 | NULL | NULL |

| 2 | 2019-02-14 | NULL | NULL |

| 3 | 2019-02-14 | NULL | NULL |

+---------+------------+---------------+---------------+

3 rows in set (0.00 sec)

**Warning**: if you're using WHERE clause to filter results, you **cannot** window alias.

SET @now = "2019-02-14";

SELECT

user\_id

,ts

,DATEDIFF(@now, LAG(ts, 1) OVER w) AS day\_from\_pre1

,DATEDIFF(@now, LAG(ts, 2) OVER w) AS day\_from\_pre2

FROM Login

WINDOW w AS (PARTITION BY user\_id ORDER BY ts)

WHERE ts = @now;

ERROR 1064 (42000): You have an error in your SQL syntax; check the manual that corresponds to your MySQL server version for the right syntax to use near 'WHERE ts = @now' at line 8

**Warning**: similarly, you cannot use window function in WHERE clause. Instead, place it in a temporary table.

SET @now = "2019-02-14";

SELECT

user\_id

,ts

FROM Login

WHERE DATEDIFF(@now, LAG(ts, 1) OVER (PARTITION BY user\_id ORDER BY ts)) AS day\_from\_pre1 = 1

AND DATEDIFF(@now, LAG(ts, 2) OVER (PARTITION BY user\_id ORDER BY ts)) AS day\_from\_pre2 = 2

AND ts = @now;

ERROR 1064 (42000): You have an error in your SQL syntax; check the manual that corresponds to your MySQL server version for the right syntax to use near 'AS day\_from\_pre1 = 1

The correct implementation.

SET @now = "2019-02-14";

WITH tmp AS (

SELECT

user\_id

,ts

,DATEDIFF(@now, LAG(ts, 1) OVER w) AS day\_from\_pre1

,DATEDIFF(@now, LAG(ts, 2) OVER w) AS day\_from\_pre2

FROM Login

WINDOW w AS (PARTITION BY user\_id ORDER BY ts)

)

SELECT user\_id

FROM tmp

WHERE ts = @now

AND day\_from\_pre1 = 1

AND day\_from\_pre2 = 2;

+---------+

| user\_id |

+---------+

| 1 |

+---------+

1 row in set (0.00 sec)

### Generalize to 7 Days

SET @now = "2019-02-14";

WITH tmp AS (

SELECT DISTINCT user\_id, ts FROM Login)

SELECT \*

FROM tmp AS d0

JOIN tmp AS d1

ON d0.ts = @now

AND DATEDIFF(@now, d1.ts) = 1

AND d0.user\_id = d1.user\_id

JOIN tmp AS d2

ON DATEDIFF(@now, d2.ts) = 2

AND d0.user\_id = d2.user\_id

JOIN tmp AS d3

ON DATEDIFF(@now, d3.ts) = 3

AND d0.user\_id = d3.user\_id

JOIN tmp AS d4

ON DATEDIFF(@now, d4.ts) = 4

AND d0.user\_id = d4.user\_id

JOIN tmp AS d5

ON DATEDIFF(@now, d5.ts) = 5

AND d0.user\_id = d5.user\_id

JOIN tmp AS d6

ON DATEDIFF(@now, d6.ts) = 6

AND d0.user\_id = d6.user\_id

JOIN tmp AS d7

ON DATEDIFF(@now, d7.ts) = 7

AND d0.user\_id = d7.user\_id;

SET @now = "2019-02-14";

WITH tmp AS (

SELECT

user\_id

,DATEDIFF(@now, LAG(ts, 1) OVER w) AS day\_from\_pre1

,DATEDIFF(@now, LAG(ts, 2) OVER w) AS day\_from\_pre2

,DATEDIFF(@now, LAG(ts, 3) OVER w) AS day\_from\_pre3

,DATEDIFF(@now, LAG(ts, 4) OVER w) AS day\_from\_pre4

,DATEDIFF(@now, LAG(ts, 5) OVER w) AS day\_from\_pre5

,DATEDIFF(@now, LAG(ts, 6) OVER w) AS day\_from\_pre6

,DATEDIFF(@now, LAG(ts, 7) OVER w) AS day\_from\_pre7

FROM Login

WINDOW w AS (PARTITION BY user\_id ORDER BY ts)

)

SELECT user\_id

FROM tmp

WHERE ts=@now

AND day\_from\_pre1 = 1

AND day\_from\_pre2 = 2

AND day\_from\_pre3 = 3

AND day\_from\_pre4 = 4

AND day\_from\_pre5 = 5

AND day\_from\_pre6 = 6

AND day\_from\_pre7 = 7;

# Recommending Friend

Write a query that identifies users who share similar tastes on music, e.g. listening to at least three songs on the same day, and recommend them to be connected as friends.

* Song: user\_id, song\_id, ts
* User: user\_id, friend\_id

This question tests multiple concepts and numerous edge cases. Among those are:

* Self-join.
* **De-duplication.**
* Exclusion.
* Equi-join, non equi-join.
* Aggregation.

To understand de-duplication, one needs to fully grasp the self-join mechanism, and be able to come up with simple example to keep track of the number of rows.

### Load Data

Load the database file [db.sql](https://github.com/NIteshx2/AdvancedSQL_Interview/blob/master/Interview/10_Spotify_Recommend_Friend/db.sql) to localhost MySQL. A SpotifyFriend database will be created with two tables.

mysql < db.sql -uroot -p

mysql> SELECT \* FROM Song LIMIT 5;

+----+---------+-------------------+------------+

| id | user\_id | song | ts |

+----+---------+-------------------+------------+

| 1 | Alex | Kiroro | 2019-03-17 |

| 2 | Alex | Shape of My Heart | 2019-03-17 |

| 3 | Alex | Clair de Lune | 2019-03-17 |

| 4 | Alex | The Fall | 2019-03-17 |

| 5 | Alex | Forever Young | 2019-03-17 |

+----+---------+-------------------+------------+

5 rows in set (0.00 sec)

mysql> SELECT \* FROM User;

+----+---------+-----------+

| id | user\_id | friend\_id |

+----+---------+-----------+

| 1 | Cindy | Bill |

+----+---------+-----------+

1 row in set (0.00 sec)

### Thought Process

First thing we notice is that we should **not** join with the User table, because doing so will give us pair of users who are already friends. The question explicitly asks for friend recommendation, meaning that we only want to return pair of users who are **not yet friends**.

There are several steps in building the recommendation pairs.

* Step 1: self-join song table.
* Step 2: aggregating the joined table, to count common songs listened by the same user on the same day.
* Step 3: Filter out user pairs who are already friends.

#### Step 1. Self Join

Let's first clear-up the arithmetic of inner join. If one row in left table finds N matches on the right table, N rows will be returned, with the one row from the left multipled N times to match the right.

Self-join is no different. If we apply enough condition to restrict matching, we only end up matching one row to one row, and the result contains the same number of rows as before self join.

SELECT

\*

FROM Song AS s1

WHERE s1.user\_id = 'Cindy'

AND s1.ts = '2019-03-14';

+----+---------+---------------+------------+

| id | user\_id | song | ts |

+----+---------+---------------+------------+

| 19 | Cindy | My Love | 2019-03-14 |

| 20 | Cindy | Clair de Lune | 2019-03-14 |

| 21 | Cindy | Lemon Tree | 2019-03-14 |

| 22 | Cindy | Mad World | 2019-03-14 |

+----+---------+---------------+------------+

4 rows in set (0.00 sec)

SELECT

\*

FROM Song AS s1

JOIN Song AS s2

ON s1.ts = s2.ts

AND s1.song = s2.song

WHERE s1.user\_id = 'Cindy'

AND s2.user\_id = 'Cindy'

AND s1.ts = '2019-03-14';

+----+---------+---------------+------------+----+---------+---------------+------------+

| id | user\_id | song | ts | id | user\_id | song | ts |

+----+---------+---------------+------------+----+---------+---------------+------------+

| 19 | Cindy | My Love | 2019-03-14 | 19 | Cindy | My Love | 2019-03-14 |

| 20 | Cindy | Clair de Lune | 2019-03-14 | 20 | Cindy | Clair de Lune | 2019-03-14 |

| 21 | Cindy | Lemon Tree | 2019-03-14 | 21 | Cindy | Lemon Tree | 2019-03-14 |

| 22 | Cindy | Mad World | 2019-03-14 | 22 | Cindy | Mad World | 2019-03-14 |

+----+---------+---------------+------------+----+---------+---------------+------------+

4 rows in set (0.00 sec)

In this example, we don't want to match a user to himself. So we need two equalities and one inequality in the self join: the date and song will be equal, and the user\_id must not be equal. Now let's self-join on a small scale, for Cindy and Bill on Mar-14.

SELECT

\*

FROM Song AS s1

WHERE s1.user\_id = 'Bill'

AND s1.ts = '2019-03-14';

+----+---------+------------+------------+

| id | user\_id | song | ts |

+----+---------+------------+------------+

| 10 | Bill | My Love | 2019-03-14 |

| 23 | Bill | Lemon Tree | 2019-03-14 |

| 24 | Bill | Mad World | 2019-03-14 |

| 25 | Bill | My Love | 2019-03-14 |

+----+---------+------------+------------+

4 rows in set (0.00 sec)

Can you predict the result of this join? Here is the accounting:

* Cindy and Bill listened to Lemon Tree once, resulting 1 row. The reverse direction also returns 1 row (the same goes for Mad World).
* Cindy listened to My Love once. Bill listened to it twice, resulting in 2 rows. The reverse direction also returns 2 rows.
* Cindy listened to Clair de Lune whereas Bill did not, returning 0 match.
* The total number of rows returned is (1 + 1) \* 2 + (2 + 2) = 8 rows.

SELECT

s1.user\_id

,s2.user\_id

,s2.song

,s2.ts

FROM (SELECT \* FROM Song WHERE user\_id IN ('Cindy', 'Bill') AND ts = '2019-03-14') AS s1

JOIN (SELECT \* FROM Song WHERE user\_id IN ('Cindy', 'Bill') AND ts = '2019-03-14') AS s2

ON s1.user\_id != s2.user\_id

AND s1.song = s2.song

ORDER BY s1.user\_id, s2.user\_id, s2.song;

+---------+---------+------------+------------+

| user\_id | user\_id | song | ts |

+---------+---------+------------+------------+

| Bill | Cindy | Lemon Tree | 2019-03-14 |

| Bill | Cindy | Mad World | 2019-03-14 |

| Bill | Cindy | My Love | 2019-03-14 |

| Bill | Cindy | My Love | 2019-03-14 |

| Cindy | Bill | Lemon Tree | 2019-03-14 |

| Cindy | Bill | Mad World | 2019-03-14 |

| Cindy | Bill | My Love | 2019-03-14 |

| Cindy | Bill | My Love | 2019-03-14 |

+---------+---------+------------+------------+

8 rows in set (0.00 sec)

The key revelation here is that self-join is a two-eay process. If a row on the left matches to two rows on the right, the two row from the right will appear somewhere inside the left table (because there is only one table). The result is **symmetric**.

#### Symmetry of Self-join

[A picture containing chart

Description automatically generated](https://github.com/NIteshx2/AdvancedSQL_Interview/blob/master/Interview/10_Spotify_Recommend_Friend/fig/symmetry.png)

From left to right, the graph represent the following self-join.

**Left graph: two edges**

SELECT \* FROM Song WHERE id IN (22, 24);

+----+---------+-----------+------------+

| id | user\_id | song | ts |

+----+---------+-----------+------------+

| 22 | Cindy | Mad World | 2019-03-14 |

| 24 | Bill | Mad World | 2019-03-14 |

+----+---------+-----------+------------+

WITH tmp AS (

SELECT \* FROM Song WHERE id IN (22, 24)

)

SELECT

\*

FROM tmp AS s1

JOIN tmp AS s2

ON s1.user\_id != s2.user\_id

AND s1.song = s2.song

ORDER BY s1.user\_id, s2.user\_id, s2.song;

+----+---------+-----------+------------+----+---------+-----------+------------+

| id | user\_id | song | ts | id | user\_id | song | ts |

+----+---------+-----------+------------+----+---------+-----------+------------+

| 24 | Bill | Mad World | 2019-03-14 | 22 | Cindy | Mad World | 2019-03-14 |

| 22 | Cindy | Mad World | 2019-03-14 | 24 | Bill | Mad World | 2019-03-14 |

+----+---------+-----------+------------+----+---------+-----------+------------+

**Middle graph: four edges**

SELECT \* FROM Song WHERE id IN (2, 15, 16);

+----+---------+-------------------+------------+

| id | user\_id | song | ts |

+----+---------+-------------------+------------+

| 2 | Alex | Shape of My Heart | 2019-03-17 |

| 15 | Bill | Shape of My Heart | 2019-03-17 |

| 16 | Bill | Shape of My Heart | 2019-03-17 |

+----+---------+-------------------+------------+

3 rows in set (0.00 sec)

WITH tmp AS (

SELECT \* FROM Song WHERE id IN (2, 15, 16)

)

SELECT

\*

FROM tmp AS s1

JOIN tmp AS s2

ON s1.user\_id != s2.user\_id

AND s1.song = s2.song

ORDER BY s1.user\_id, s2.user\_id, s2.song;

+----+---------+-------------------+------------+----+---------+-------------------+------------+

| id | user\_id | song | ts | id | user\_id | song | ts |

+----+---------+-------------------+------------+----+---------+-------------------+------------+

| 2 | Alex | Shape of My Heart | 2019-03-17 | 16 | Bill | Shape of My Heart | 2019-03-17 |

| 2 | Alex | Shape of My Heart | 2019-03-17 | 15 | Bill | Shape of My Heart | 2019-03-17 |

| 15 | Bill | Shape of My Heart | 2019-03-17 | 2 | Alex | Shape of My Heart | 2019-03-17 |

| 16 | Bill | Shape of My Heart | 2019-03-17 | 2 | Alex | Shape of My Heart | 2019-03-17 |

+----+---------+-------------------+------------+----+---------+-------------------+------------+

4 rows in set (0.00 sec)

**Right graph: eight edges**

SELECT \* FROM Song WHERE id IN (12, 13, 14, 15);

+----+---------+-------------------+------------+

| id | user\_id | song | ts |

+----+---------+-------------------+------------+

| 12 | Alex | Shape of My Heart | 2019-03-17 |

| 13 | Alex | Shape of My Heart | 2019-03-17 |

| 14 | Bill | Shape of My Heart | 2019-03-17 |

| 15 | Bill | Shape of My Heart | 2019-03-17 |

+----+---------+-------------------+------------+

4 rows in set (0.01 sec)

WITH tmp AS (

SELECT \* FROM Song WHERE id IN (12, 13, 14, 15)

)

SELECT

\*

FROM tmp AS s1

JOIN tmp AS s2

ON s1.user\_id != s2.user\_id

AND s1.song = s2.song

ORDER BY s1.user\_id, s2.user\_id, s2.song;

+----+---------+-------------------+------------+----+---------+-------------------+------------+

| id | user\_id | song | ts | id | user\_id | song | ts |

+----+---------+-------------------+------------+----+---------+-------------------+------------+

| 12 | Alex | Shape of My Heart | 2019-03-17 | 14 | Bill | Shape of My Heart | 2019-03-17 |

| 13 | Alex | Shape of My Heart | 2019-03-17 | 14 | Bill | Shape of My Heart | 2019-03-17 |

| 12 | Alex | Shape of My Heart | 2019-03-17 | 15 | Bill | Shape of My Heart | 2019-03-17 |

| 13 | Alex | Shape of My Heart | 2019-03-17 | 15 | Bill | Shape of My Heart | 2019-03-17 |

| 15 | Bill | Shape of My Heart | 2019-03-17 | 13 | Alex | Shape of My Heart | 2019-03-17 |

| 14 | Bill | Shape of My Heart | 2019-03-17 | 12 | Alex | Shape of My Heart | 2019-03-17 |

| 15 | Bill | Shape of My Heart | 2019-03-17 | 12 | Alex | Shape of My Heart | 2019-03-17 |

| 14 | Bill | Shape of My Heart | 2019-03-17 | 13 | Alex | Shape of My Heart | 2019-03-17 |

+----+---------+-------------------+------------+----+---------+-------------------+------------+

8 rows in set (0.00 sec)

#### Step 2. Aggregation

Identify that there are three columns to aggregate over: user\_id pair and date.

SELECT

s1.user\_id

,s2.user\_id

,COUNT(DISTINCT s2.song)

,s2.ts

FROM (SELECT \* FROM Song WHERE user\_id IN ('Cindy', 'Bill') AND ts = '2019-03-14') AS s1

JOIN (SELECT \* FROM Song WHERE user\_id IN ('Cindy', 'Bill') AND ts = '2019-03-14') AS s2

ON s1.user\_id != s2.user\_id

AND s1.ts = s2.ts

AND s1.song = s2.song

GROUP BY s1.user\_id, s2.user\_id, s2.ts;

+---------+---------+-------------------------+------------+

| user\_id | user\_id | COUNT(DISTINCT s2.song) | ts |

+---------+---------+-------------------------+------------+

| Bill | Cindy | 3 | 2019-03-14 |

| Cindy | Bill | 3 | 2019-03-14 |

+---------+---------+-------------------------+------------+

2 rows in set (0.00 sec)

It's clear that the user\_id pair is counted both way, and the count is correct and identical! This is the most confusing part of thie question.

Now we can replace the truncated table with the full table.

SELECT

s1.user\_id

,s2.user\_id

,COUNT(DISTINCT s2.song) AS common\_song

,s2.ts

FROM Song AS s1

JOIN Song AS s2

ON s1.user\_id != s2.user\_id

AND s1.ts = s2.ts

AND s1.song = s2.song

GROUP BY s1.user\_id, s2.user\_id, s2.ts

ORDER BY s2.ts, common\_song;

+---------+---------+-------------+------------+

| user\_id | user\_id | common\_song | ts |

+---------+---------+-------------+------------+

| Bill | Cindy | 3 | 2019-03-14 |

| Cindy | Bill | 3 | 2019-03-14 |

| Bill | Cindy | 1 | 2019-03-17 |

| Cindy | Bill | 1 | 2019-03-17 |

| Alex | Cindy | 2 | 2019-03-17 |

| Cindy | Alex | 2 | 2019-03-17 |

| Bill | Alex | 4 | 2019-03-17 |

| Alex | Bill | 4 | 2019-03-17 |

+---------+---------+-------------+------------+

8 rows in set (0.01 sec)

#### Step 3. Filtering

First we need to filter user\_id pair that have fewer than 3 common song on any day.

SELECT

s1.user\_id

,s2.user\_id

,COUNT(DISTINCT s2.song) AS common\_song

,s2.ts

FROM Song AS s1

JOIN Song AS s2

ON s1.user\_id != s2.user\_id

AND s1.ts = s2.ts

AND s1.song = s2.song

GROUP BY s1.user\_id, s2.user\_id, s2.ts

HAVING common\_song >= 3;

+---------+---------+-------------+------------+

| user\_id | user\_id | common\_song | ts |

+---------+---------+-------------+------------+

| Alex | Bill | 4 | 2019-03-17 |

| Bill | Alex | 4 | 2019-03-17 |

| Bill | Cindy | 3 | 2019-03-14 |

| Cindy | Bill | 3 | 2019-03-14 |

+---------+---------+-------------+------------+

4 rows in set (0.00 sec)

Next, we need to filter user\_id pairs that are already friend. Because the Friend table accounts for one way edge, we need to UNION with the opposite direction.

SELECT

s1.user\_id

,s2.user\_id AS recommended

FROM Song AS s1

JOIN Song AS s2

ON s1.user\_id != s2.user\_id

AND s1.ts = s2.ts

AND s1.song = s2.song

WHERE (s1.user\_id, s2.user\_id) NOT IN (

SELECT user\_id, friend\_id FROM User

UNION

SELECT friend\_id, user\_id FROM User

)

GROUP BY s1.user\_id, s2.user\_id, s2.ts

HAVING COUNT(DISTINCT s2.song) >= 3;

+---------+-------------+

| user\_id | recommended |

+---------+-------------+

| Alex | Bill |

| Bill | Alex |

+---------+-------------+

2 rows in set (0.00 sec)

We can probably stop here, because the recommendation will be sent to both nodes of the same edge. For example, just take a single column, and recommend to them the user\_id from the second column.

#### Parting Thought

Do not directly start writing code. There are so many ways things can go wrong that you're bound to run out of time correcting mistakes. Work with simple examples on the whiteboard to get the general direction correct, don't run into crazy mistakes such as joining User table with Song table (automatic fail). Identify some edge cases such as user listens to the same song multiple times a day. Discuss how to handle NULL (though in this song table, both song and user\_id are foreign keys, and by referential integrity, must have parent key).