CODE CHALLENGE: MULTIPLE TABLES

plans
A table describing the various monthly subscription plans that Songify offers

Column	Description
id	A unique identifier for the plan
price	The monthly cost of the plan
description	A description of the plan

users
A table describing both free and paid users of Songify

Column	Description
id	A unique identifier for each user
first_name	The first name of the user
last_name	The last name of the user
age	The age name of the user
gender	The gender name of the user

premium_users
A table describing only the paid users of Songify

Column	Description
user_id	A unique identifier for each user Matches id in users
membership_plan_id	An ID for which payment plan that user is on Matches id in plans
purchase_date	The date when the user purchased their premium plan
cancel_date	The date when the user canceled their plan (which can be `NULL` if they haven't canceled yet)

songs
A list of all songs available on Songify

Column	Description
id	A unique identifier for each song
title	The title of the song
artist	The artist who recorded the song
year	The year that the song was released

plays
A table describing the songs played by each user on Songify

Column	Description
user_id	A unique identifier for each user Matches id in users
song_id	An ID for which payment plan that user is on Matches id in songs
play_date	The date when the user played this song
play_hour	The hour of day (0 - 23) when the user played this song

months
A table with months in the year

Column	Description
months	the first date of a month

Credit: codeacademy
Compiled by: Sumaia Parveen Shupti

Songify Introduction

Welcome to Code Challenge: Multiple Tables!

In this Code Challenge, you'll be performing analysis for Songify, a fictional music streaming service. Songify has a "freemium" model, meaning that it offers both a free product and a premium paid product.

You'll be working with six tables:

- `plans`
- `users`
- `premium_users`
- `songs`
- `months`
- `plays`
-

Code Challenge 1

For this challenge, you'll use the following tables:

`plans`

Column	Description
id	A unique identifier for the plan
price	The monthly cost of the plan
description	A description of the plan

`premium_users`

Column	Description
user_id	A unique identifier for the user
membership_plan_id	An ID for the user's payment plan (matches `plans.id`)
purchase_date	Date when the user purchased their premium plan
cancel_date	Date when the user canceled (NULL if they haven't)

Click here for the table descriptions.

1. Let's see which plans are used by which premium members!
The column `membership_plan_id` in `premium_users` should match the column `id` in `plans`.
Join `plans` and `premium_users` and select:

- `user_id` from `premium_users`
- `description` from `plans`

(Be sure to select the columns in this order)

```
select premium_users.user_id, plans.description
from premium_users
join plans
on plans.id = premium_users.membership_plan_id
limit 10;
```

Code Challenge 2

For this exercise, you'll use the following tables:

`songs`

Column	Description
`id`	A unique identifier for each song
`title`	The title of the song
`artist`	The artist who recorded the song
`year`	The year that the song was released

`plays`

Column	Description
`user_id`	A unique identifier for each user
`song_id`	An ID for which song was played (matches `songs.id`)
`play_date`	The date when the user played this song
`play_hour`	The hour when the user played this song (0-23)

Click here for the table descriptions.

1. Let's see the titles of songs that were played by each user! The column `song_id` in `plays` should match the column `id` in `songs`. Join `plays` to `songs` and select:
   - `user_id` from `plays`
   - `play_date` from `plays`
   - `title` from `songs`

(Be sure to select the columns in this order)

```
select plays.user_id, plays.play_date, songs.title
from plays
join songs
on songs.id = plays.song_id
limit 10;
```

Code Challenge 3

For this challenge, you'll use the following tables:

`users`

Column	Description
`id`	A unique identifier for each user
`first_name`	The first name of the user
`last_name`	The last name of the user
`age`	The age name of the user
`gender`	The gender name of the user

`premium_users`

Column	Description
`user_id`	A unique identifier for each user
`membership_plan_id`	An ID for the user's payment plan (matches `plans.id`)
`purchase_date`	Date when the user purchased their premium plan
`cancel_date`	Date when the user canceled (`NULL` if they haven't)

Click here for the table descriptions.

1. Which users *aren't* premium users?
Use a `LEFT JOIN` to combine `users` and `premium_users` and select `id` from `users`. The column `id` in `users` should match the column `user_id` in `premium_users`. Use a `WHERE` clause to limit the results to users where `premium_users.user_id IS NULL`. This will remove premium users and leave you with only free users.

```
select users.id
 from users
 left join premium_users
 on premium_users.user_id  = users.id
 where premium_users.user_id is null;
```

Code Challenge 4

We've used a `WITH` statement to create two temporary tables:

- `january` contains all song plays from January 2017
- `february` contains all song plays from February 2017

If you need help, check out this Reference Guide to multiple tables in SQL.

1. Use a left join to combine `january` and `february` on `user_id` and select `user_id` from `january`. Add the following `WHERE` statement to find which users played songs in January, but not February:

```
WHERE february.user_id IS NULL
```

```
WITH january AS (
  SELECT *
  FROM plays
  WHERE strftime("%m", play_date) = '01'
),
february AS (
  SELECT *
  FROM plays
  WHERE strftime("%m", play_date) = '02'

)
select january.user_id
from january
left join february
on february.user_id = january.user_id
where february.user_id is null;
```

Code Challenge 5

For this challenge, you'll use the following tables:

`months`

Column	Description
`months`	The first date of each month of the year

`premium_users`

Column	Description
`user_id`	A unique identifier for the user
`plan_id`	An ID for the user's payment plan (matches `plans.id`)
`purchase_date`	Date when the user purchased their premium plan
`cancel_date`	Date when the user canceled (`NULL` if they haven't)

Click here for the table descriptions.

1. For each month in `months`, we want to know if each user in `premium_users` was active or canceled. Cross join `months` and `premium_users` and select:

- `user_id` from `premium_users`
- `purchase_date` from `premium_users`
- `cancel_date` from `premium_users`
- `months` from `months`

(Be sure to select the columns in this order)

```
select user_id, purchase_date, cancel_date, months
 from premium_users
 cross join months;
```

Code Challenge 6

1. Replace the `SELECT` statement in your `CROSS JOIN` with the following statement:

```
SELECT premium_users.user_id,
  months.months,
  CASE
    WHEN (
      premium_users.purchase_date <= months.months
      )
      AND
      (
        premium_users.cancel_date >= months.months
        OR
        premium_users.cancel_date IS NULL
      )
    THEN 'active'
    ELSE 'not_active'
  END AS 'status'
```

This will tell us if a particular user is `'active'` or `'not_active'` each month.

```
SELECT premium_users.user_id,
  months.months,
  CASE
  WHEN (premium_users.purchase_date <= months.months)
      AND
       (premium_users.cancel_date >= months.months
      OR
      premium_users.cancel_date IS NULL)
    THEN 'active'
    ELSE 'not_active'
  END AS 'status'
FROM premium_users
CROSS JOIN months
limit 10;
```

Code Challenge 7

Songify has added some new songs to their catalog. Combine `songs` and `bonus_songs` using `UNION` and select all columns from the result. Since the `songs` table is so big, just look at a sample by `LIMIT`ing the results to `10` rows.

```
select *
from songs
union
select *
from bonus_songs
limit 10;
```

Code Challenge 8

Besides stacking one table on top of another, we can also use `UNION` to quickly make a "mini" dataset:

```
SELECT '2017-01-01' AS 'month'
UNION
SELECT '2017-02-01' AS 'month'
```

will produce:

month
2017-01-01
2017-02-01

1. Modify the query in **test.sqlite**: Add a third `UNION`/`SELECT` so that the result contains `2017-03-01`.

```
SELECT '2017-01-01' as month
UNION
SELECT '2017-02-01' as month
UNION
SELECT '2017-03-01' as month;
```

Code Challenge 9

The following exercise uses the Songify tables explained before. You can look up the schema of those tables here.

1. The following query will give us the number of times that each song was played:

```
SELECT song_id,
   COUNT(*) AS 'times_played'
FROM plays
GROUP BY song_id;
```

Use a `WITH` statement to alias this code as `play_count`.

Join `play_count` with `songs` and select (in this order):

- `songs` table's `title` column
- `songs` table's `artist` column
- `play_count`'s `times_played` column
-

Remember that `play_count.song_id` will match `songs.id`.

```
with play_count as (SELECT song_id, COUNT(*) AS 'times_played'
FROM plays
GROUP BY 1)
select songs.title, songs.artist, play_count.times_played
from songs
join play_count
on play_count.song_id = songs.id
limit 10;
```