Week 6: Unstructured Data

Advanced SQL Concepts

Contents

We will be using the `yelp` database for this lab.

Unstructured Data

Often when we work with data in practice, it is stored in its raw format. For example, APIs usually return data in the form of JSON (Javascript Object Notation).

Often when we are aggregating data, we are aggregating across multiple dimensions. For example, if I want to see the total order value for each customer for each product category, I would want to write something like this:

```
SELECT name, attributes
FROM businesses LIMIT 5;
```

You'll notice a field called `attributes`, which appears to have data that looks similar to a dictionary object in languages such as Python.

If we look closely at the values for this field, we see a data structure:

```
{
   "GoodForKids":"False",
   "BusinessAcceptsCreditCards":"True",
   "BikeParking":"False",
   "Alcohol":"u'full_bar'",
   "RestaurantsReservations":"True",
   "OutdoorSeating":"False",
   "RestaurantsTakeOut":"False",
   "RestaurantsAttire":"'casual'",
   "RestaurantsPriceRange2":"2",
   "Caters":"False",
   "RestaurantsDelivery":"False",
   "RestaurantsGoodForGroups":"True",
   "WiFi":"u'no'"
   "NoiseLevel":"u'quiet'",
   "CoatCheck":"True",
   "HasTV":"True"
}
```

These contain attributes about the restaurant packed together in a dictionary-like object called **JSON (Javascript Object Notation)**. JSON is a format for storing and transporting data that has now become common on the internet because it is relatively human-readable and "self-describing". It is commonly used as the language format of choice when sending responses to and from an API server.

Unpacking JSON

To get the value of the `HasTV` attribute, for instance, we can use the following notation:

```
SELECT name                         business_name,
       attributes::JSON ->> 'HasTV' has_tv
FROM businesses
LIMIT 5;
```

The result we get is below:

business_name	has_tv
Sushime	False
El Toro	NA
Fabric Boutique	NA
Riviera Coffee Company	NA
Captain's Galley Seafood	False

Let's unpack what just happened. We first

- Cast the `attributes` field to a `JSON` data type. It was originally a `TEXT` data type.
- Then we use the unpack operator `->>` to get the `HasTV` field within the `attributes` JSON object.
- We return this field as a separate column called `has_tv`. For rows that do not have a `HasTV` field, we return `NULL`.

1. Using the unpack operator, return the `BusinessAcceptsCreditCards` field of the `attributes` JSON. Name the returned field `accepts_credit_cards`.

```
SELECT business_id,
       name,
       city,
       state,
       attributes::JSON ->> 'BusinessAcceptsCreditCards' accepts_credit_cards
FROM businesses
LIMIT 10;
```

business_id	name	city	state	accepts_credit_cards
Vj-0b1zQVwkecyfuhd0E0w	Sushime	Calgary	AB	NA
kBgpdaS9joBXioj89zo1QA	El Toro	Urbana	IL	True
C5y_vNSoncFF-_pn0kuVeQ	Fabric Boutique	Las Vegas	NV	True
FxqSR54O6m_HtgD18W8ciw	Riviera Coffee Company	Chandler	AZ	True
OAwpyOX2rISg7MEuHLd_6A	Captain's Galley Seafood	Matthews	NC	True
qhuuWs1eshxEHhTSfGV2rg	Sapporo Asian Cuisine & Sushi Bar	Monroeville	PA	True
0Em_6So49P5c0j1bJABo3g	Jos. A. Bank	Pittsburgh	PA	NA
p0iEUamJVp_QpaheE-Nz_g	South Point Hotel, Casino & Spa	Las Vegas	NV	True
7K9EGbodeoDoOzQvkNPoAw	Cinéma Cineplex Odeon Brossard et VIP	Brossard	QC	NA
PR76mJ-p-K0I-PAi3nVC_A	Burlington Coat Factory	Charlotte	NC	True

Notice that the values being returned in the `accepts_credit_cards` field are NOT booleans - they are text values. Also notice that if a restaurant does not have this particular field in their attributes, it is NULL (in this PDF, represented as NA).

COALESCE

Sometimes, we do not want null results in our result set or computation. NULLs can be undesirable for many reasons:

- They are not client-facing or easy to understand for non-technical stakeholders (try explaining to an executive why a value is NULL)
- They self-propagate during aggregation or mathematical operations. For example, if you add perform the operation NULL + 2 + 3 + 0:

```
SELECT NULL + 2 + 3 + 0 as result
```

result
NA

Therefore, entire queries are often nulled-out if even one value is NULL. We want to set a **default value** for a value that returns a NULL. In our case, we want to have all NULL values equal the text value False.

We'll use the COALESCE function to implement this type of default fallback logic:

The COALESCE function takes in a variable number of arguments, and **returns the first non-null value** starting from the left and moving right in the position of arguments. Notice what happens when we write the following:

```
SELECT COALESCE(4, NULL, 2) col1,
       COALESCE(NULL, 1, 2) col2,
       COALESCE(NULL, NULL, 'final_value') col3,
       COALESCE(NULL, NULL, NULL) col4
```

Check out the results of each of these columns.

col1	col2	col3	col4
4	1	final_value	NA

For the first column, we return 4, since the first non-null value moving from left to right is 4. For the second column, since the first value is NULL, we check the second column, and receive 1, which is non-null.

```
SELECT name,
       city,
       state,
       COALESCE(attributes::JSON ->> 'BusinessAcceptsCreditCards', 'False') accepts_credit_cards
FROM businesses
LIMIT 10;
```

Now the results look much better:

name	city	state	accepts_credit_cards
Sushime	Calgary	AB	False
El Toro	Urbana	IL	True
Fabric Boutique	Las Vegas	NV	True
Riviera Coffee Company	Chandler	AZ	True
Captain’s Galley Seafood	Matthews	NC	True
Sapporo Asian Cuisine & Sushi Bar	Monroeville	PA	True
Jos. A. Bank	Pittsburgh	PA	False
South Point Hotel, Casino & Spa	Las Vegas	NV	True
Cinéma Cineplex Odeon Brossard et VIP	Brossard	QC	False
Burlington Coat Factory	Charlotte	NC	True

Nested JSON

Let’s look now at the `BusinessParking` field of the `attributes` JSON object:

```
SELECT attributes::JSON ->> 'BusinessParking' business_parking
FROM businesses
LIMIT 10;
```

Notice that it itself is a valid JSON object.

business_parking
{’garage’: False, ’street’: False, ’validated’: False, ’lot’: False, ’valet’: False}
{’garage’: False, ’street’: False, ’validated’: False, ’lot’: True, ’valet’: False}
{’garage’: False, ’street’: False, ’validated’: False, ’lot’: True, ’valet’: False}
NA
{’garage’: False, ’street’: False, ’validated’: False, ’lot’: False, ’valet’: False}
{’garage’: False, ’street’: False, ’validated’: False, ’lot’: False, ’valet’: False}
NA
{’garage’: True, ’street’: False, ’validated’: False, ’lot’: False, ’valet’: False}
NA
{’garage’: False, ’street’: False, ’validated’: False, ’lot’: False, ’valet’: False}

However, there’s an important rule about JSON to be aware of: strings (text) need to be in double quotes. A JSON object is not the same as a Python dictionary object.

Inside of our `BusinessParking` field value, we see entries with only single quotes, and values with neither single nor double quotes. We need to convert this string into proper JSON format by replacing all instances of `'` with `"` (single quotes with double quotes). And any values that do not have quotes around them - like `False`, or `True`, need to be wrapped in double quotes.

```
SELECT *,
       (
           REPLACE(
                   REPLACE(
                           REPLACE(
                                   REPLACE(
                                           (attributes::JSON ->> 'BusinessParking'),
                                           '''', '"'), -- replace single quotes w/ double quotes
                                   'False', '"False"'), -- replace False with "False"
                           'True', '"True"'),  -- replace True with "True"
               'None','"None"'))::JSON business_parking_json
FROM businesses
LIMIT 5;
```

This query performs the following:

- Casts `attributes` to a JSON data type and unpacks the `BusinessParking` field, which is a text value.
- Replaces all single quotes with double quotes in the text value returned. For example `'parking'` becomes `"parking"`.
- Replaces all instances of `True` or `False` with `"True"` and `"False"`, respectively.

This now prepares the text to be usable as valid JSON.

This is already quite a complicated query, so let's save it as a view. We'll call the view `parsed_businesses` and we'll save it to our personal schemas.

```
CREATE OR REPLACE VIEW ychen220.parsed_businesses AS
SELECT *,
       (
           REPLACE(
                   REPLACE(
                           REPLACE(
                                   REPLACE(
                                           (attributes::JSON ->> 'BusinessParking'),
                        '''', '"'), -- replace single quotes w/ double quotes
                    'False', '"False"'), -- replace False with "False"
                  'True', '"True"'),  -- replace True with "True"
               'None','"None"'))::JSON business_parking_json
FROM businesses;
```

We can then use this view to query for parking information about each of the restaurants.

2. What percentage of the top 50 restaurants/businesses have parking lots for customers?

```
SELECT COUNT(*) * 1.0 / -- get the total count of businesses with parking lots
       (SELECT COUNT(*) FROM ychen220.parsed_businesses)
       -- use a subquery to get the total count of all the businesses
    AS percentage_with_parking_lots
FROM ychen220.parsed_businesses
WHERE business_parking_json ->> 'lot' = 'True'
```

percentage_with_parking_lots
0.3

3. **YOUR TURN** Parse the `Ambience` key of `attributes` JSON. Identify what percentage of the top 50 restaurants/establishments are considered casual. How to approach this problem:

- convert the `Ambience` key into valid JSON by replacing single quotes with double quotes, text with no quotes with double quotes, etc.
- then parse the `casual` key from this new JSON object and count the number that are `True`.

percentage_casual
0.12

Arrays

Suppose the marketing manager of your company wants to find out the most popular categories for the different businesses on Yelp. She says to look at the `categories` field.

We often have a list of values in a text field that we want to convert into a formal array data structure. For example, let's take a look at the `categories` field of the `businesses` table.

```
SELECT categories
FROM businesses
LIMIT 5;
```

We can see that it is a text field, split by commas.

categories
Japanese, Sushi Bars, Restaurants
Mexican, Restaurants
Arts & Crafts, Fabric Stores, Shopping
Restaurants, Coffee & Tea, Food, Professional Services, Coffee Roasteries, Wholesalers
Restaurants, Seafood

Although this is useful, it's not the exact proper data structure to work with. For example, the `Riviera Coffee Company` business in Chandler, Arizona, is tagged with the category `Food`. If we want to find other restaurants that share this tag, we might write something like this: '

```
SELECT categories
FROM businesses
WHERE categories ILIKE '%Food%'
```

However, this would return `Seafood`, `American Food`, `Chinese Food`, etc.

Let's split the text field into an array of strings using `string_to_array`:

```
SELECT name, categories, string_to_array(categories, ', ') AS category_array
FROM businesses
LIMIT 5;
```

Inspect the difference between `categories` and `category_array`:

name	categories	category_array
Sushime	Japanese, Sushi Bars, Restaurants	{Japanese,"Sushi Bars",Restaurants}
El Toro	Mexican, Restaurants	{Mexican,Restaurants}
Fabric Boutique	Arts & Crafts, Fabric Stores, Shopping	{"Arts & Crafts","Fabric Stores",Shopping}
Riviera Coffee Company	Restaurants, Coffee & Tea, Food, Professional Services, Coffee Roasteries, Wholesalers	{Restaurants,"Coffee & Tea",Food,"Professional Services","Coffee Roasteries",Wholesalers}
Captain's Galley Seafood	Restaurants, Seafood	{Restaurants,Seafood}

We'll create a view to save this logic:

```
CREATE OR REPLACE VIEW ychen220.parsed_categories AS
SELECT business_id, name, categories, string_to_array(categories, ', ') AS category_array
FROM businesses;
```

Now, we'll use this `category_array` to find most common categories across the top 50 businesses:

```
SELECT category, COUNT(DISTINCT business_id) AS num_businesses
FROM (
-- unnest takes an array and explodes it out to one row for each element of array
         SELECT business_id, unnest(category_array) AS category
         FROM ychen220.parsed_categories) t1
GROUP BY 1
```

```
ORDER BY 2 DESC
LIMIT 5; -- take the top 5 categories
```

You can see that `Restaurant` is the most popular category:

category	num_businesses
Restaurants	19
Shopping	10
Local Services	6
Food	5
Health & Medical	5

4. **YOUR TURN** Provide the list of top 3 businesses for `Shopping`. Use the average rating stars of `reviews` to sort your businesses and rank them.

Steps:

- First get all restaurants that have the `Shopping` category.
- Then join to `reviews`
- Finally, `GROUP BY` the restaurant name and get the average star rating and number of reviews.

You should be able to see that `Fabric Boutique` has the highest star ratings.

name	average_rating	num_ratings
Fabric Boutique	4.800000	5
The Estate Watch & Jewelry Company	4.426667	75
Quick Mobile Repair	4.012987	77

5. **YOUR TURN** We want to provide a special promotion for restaurants reaching their 100th check in. List the date that a business received its 100th check in.

Steps:

- Use the `checkins` table's `date` field and parse into an array of dates (hint - use `string_to_array` and `unnest`).
- Then use a window function to get the running count of check-in dates.
- Filter for the 100th check-in count for each business.

The result you should receive is below:

business_id	checkin_date	checkin_number
0bF6jv97Z6VbruRpepUOcw	2014-04-20 01:42:55	100
7K9EGbodeoDoOzQvkNPoAw	2019-05-10 00:27:34	100
8yZ_nBgRC3CT1RFJC6KhXA	2014-08-11 17:01:58	100
-DUbOEoenPbeJu35e9Ukrg	2014-01-01 17:52:57	100
EUDKL2TNV1kpeUQm_QZvCQ	2016-06-29 11:30:55	100
p0iEUamJVp_QpaheE-Nz_g	2010-08-21 08:06:19	100

6. **YOUR TURN** Find the average number of reviews for each user's Yelp friends. Use the `friends` field. Only consider users with under 4 reviews.

Hints:

- First, create a result set that contains `user_id` and `friend_id`, like the below, where each friend of a user is listed in a new row:

user_id	friend_id
vJgybdPuxTn8uaHmmyLLTw	1VHycZatAvC-8JqTFfFM8w
vJgybdPuxTn8uaHmmyLLTw	xPPlOHXScld0dunZ2-Cg5w
vJgybdPuxTn8uaHmmyLLTw	SEDjISzOL32w6GAm_dM3bw
vJgybdPuxTn8uaHmmyLLTw	PtzWJkQD2Vl4iyHOqHiTrg
vJgybdPuxTn8uaHmmyLLTw	SxXCFT0dwJDYRUC-OGk1eg

- Then, you'll want to perform a join from this result set (make it a subquery or CTE) back to your `users` table. Think carefully about what columns you want to be joining. Once you've performed your join, you can then perform your `GROUP BY` to get the `AVG(review_count)`.

Your expected final result:

user_id	avg_reviews_of_friends
l0mi5p-A5GBLiJECt6AIOA	288
LGkHW3X0PToy_Fepc2PBww	217
TYEJF4E-9BgHvc_YT24DTg	54
QIYgh1kbwCUOORpOPfJOOQ	28
-3It-sACFwohtprscSIKkg	28
6HH9wB-hE-ndIlBsu438oQ	21
CiaMoLyHegAbffRn5Qh-6g	12
QXMlOp0EiRWRi5XAebAJUw	5