W4111.001-Introduction to Databases
Spring 2021
Databases Project Part 4 submission
Due April 7 at 11:59 PM
Group Number 31

Group Members

md3420 -- Matthew Duran
ms5767 -- Meghan Shah

UNI tagged to the postgresql database is **md3420**

- In the database tagged by the above uni, we have our original schema from parts 1-3 along with the new object relational features that we have implemented
- This is the command we type to access the database from our vm instance

Psql -U md3420 -h 34.73.36.248 -d project1

We decided to implement the following features:

1) Text attribute with text search queries
2) Array attribute with array based queries
3) Triggers and functions

Note: we will also attach a **group31.sql file** with all of the code we wrote to fulfil these specifications.

Part 1: Text Attributes

In order to utilize the text attribute and full-text search, we decided to create a table of restaurant reviews with review as a text attribute. We made a new table for this, with a cust_id attribute that was a foreign key, referencing the customer table, and a rev_id primary key to go with each unique review. We also included a date attribute per review. In inserting items into the table, we inserted the restaurant reviews as blocks of text, surrounded by single quotes. In the full-text search portion, we then utilized tsvectors to translate our text attributes into lexemes prior to searching them. We are attaching some of the reviews that we used here, as the display in postgres is somewhat hard to decipher:

Review 1: Highly recommend this place! They make a mean grapefruit mimosa and their custard french toast will make its way into your dreams. No, seriously. I'm haunted. If only it wasn't so overbooked, I'd be a regular. Don't sleep on the salmon toast either. They have it all!

Review 2: Okay, this place doesn't have bad food. They actually pull off the whole fusion thing, which isn't easy to do, especially for brunch. I'd get the crab scramble, because that's pretty hard to get elsewhere at this quality. But beware, the wait times can be so long on weekends. And one of the servers was a little rude.

Review 3: lemon ricotta pancakes!!! I cannot recommend them more! Go to this restaurant and order them for brunch, you will NOT regret it.

Review 4: Overcrowded, and too loud! Wanted to come here for a nice date, but it took so long to get our food. We got the soft scrambled eggs, which were not worth the money, but the buttermilk biscuits were the only redeeming factor of this place.

Review 5: Eating this food was the taste equivalent of hearing nails on a chalkboard. Painful. Would you rather eat dirt or eat at this restaurant? Easy, dirt. That is all. Save yourself.

Text Queries:

1) We started out with simple queries in order to see which reviews contained a specific word - if for example a customer was looking for all the reviews that were about pancakes:
    - Reference query 1 in 'text attributes' section of **group31.sql**
2) Next we decided to try to use the boolean based full-text search operators to get us closer to some type of sentiment analysis - to gauge which reviews were positive versus negative. In order to do that, we combined the boolean operators OR, AND and NOT to try to look for certain positive sentiment words, and the lack of certain negative ones. We ran into a number of limitations here, which we thought we'd document below.
    - Reference query 2 in 'text attributes' section of **group31.sql**
        - This query allowed us to output the reviews that contained words like "recommend" or "good", but also didn't have the world "beware." This worked for our purposes.
    - Reference query 3 in 'text attributes' section of **group31.sql**
        - However, if we were to repeat this query with 'good' instead of 'great', it would no longer provide the desired results. The query result would now include a very negative review, only reported because it has the word 'good' in it, when it is in fact 'not good'.

- These are some of the limitations of trying to implement sentiment analysis in SQL. It would have been nice to do sentiment analysis in a more systematic way, but I think that is a problem more suited to a python implementation using a dictionary of words with sentiment values.

Note: check query results on postgres, unfortunately the result table doesn't output in a friendly format

Part 2: Array Attributes

In order to utilize the array attribute type, we decided to create a new table that would be a rough way to track employee performance, and employee satisfaction ratings of their experience at the company. The conceptual goal was to create a tool for the restaurant to keep track of its employees' performance changes/trends over time. Further, they could get a sense of their employees' happiness with working at the restaurant in order to review their own work-environment. We also thought this combination could potentially be useful to see potential trends between employee satisfaction and performance. The table included employee_id (which was the primary key, and also referenced the employee table), along with a manager_id to keep track of which employees were reviewed by which managers, and two array attribute columns: one which stored employee performance reviews per month, as a rating from 1-10 (ex: [2,3,5,6,10,10,9,10,9,10,10,10]), and the other which stored employee self-reports of their satisfaction in the same form. We then inserted this performance data for approximately 20 employees.

Now that we have the performance_reviews table with the array attributes, let's talk about the query we wrote to manipulate the array data. In the emp_performance_review_ rating_bymonth attribute, we have an array of 12 integers, which correspond to their performance rating for a given month (where indexes correspond to Jan through December). We wanted to write a query to determine whether a worker was performing better in the second half of the year as opposed to the first half of the year. If this were the case, then that worker would be in line for a raise and if the opposite occurred the employee would be in line for a pay cut. This query was super complicated (see the two heavily nested queries in the 'array attributes' code section of **group31.sql**), but ends up manipulating the underlying array structure well. Since both queries are mirror images, except for flipping a sign in the where clause, we will describe the one where we determine which workers are in line for pay raises.

performance_review table

employee_id	manager_id	emp_performance_rating_bymonth	emp_satisfaction_rating_bymonth	month_array
16096	36708	{6,7,7,8,9,9,9,10,9,10,10,10}	{10,10,9,8,10,10,10,10,10,10,10,10}	{Jan,Feb,Mar,Apr,May,Jun,Jul,Aug,Sep,Oct,Nov,Dec}
10772	18230	{6,7,7,8,9,9,9,6,9,8,8,9}	{5,3,9,8,7,10,10,10,10,10,8,10}	{Jan,Feb,Mar,Apr,May,Jun,Jul,Aug,Sep,Oct,Nov,Dec}
12704	18230	{10,9,7,8,5,9,9,6,9,10,10,10}	{10,10,9,8,10,10,10,10,5,6,6,8}	{Jan,Feb,Mar,Apr,May,Jun,Jul,Aug,Sep,Oct,Nov,Dec}
24141	48049	{5,5,7,8,9,9,10,10,9,9,10,10}	{6,6,9,8,9,10,10,9,10,9,9,9}	{Jan,Feb,Mar,Apr,May,Jun,Jul,Aug,Sep,Oct,Nov,Dec}
16417	48049	{6,7,9,8,9,6,9,10,6,10,9,10}	{10,10,9,10,10,10,10,10,10,10,10,10}	{Jan,Feb,Mar,Apr,May,Jun,Jul,Aug,Sep,Oct,Nov,Dec}
33542	48049	{6,7,7,8,8,9,9,10,9,10,5,10}	{10,8,9,8,7,10,10,6,10,6,10,10}	{Jan,Feb,Mar,Apr,May,Jun,Jul,Aug,Sep,Oct,Nov,Dec}
45355	48049	{6,8,5,8,4,9,9,10,6,10,8,10}	{10,10,9,8,9,10,9,10,10,10,8,10}	{Jan,Feb,Mar,Apr,May,Jun,Jul,Aug,Sep,Oct,Nov,Dec}
38707	48049	{6,7,7,9,9,9,9,10,9,9,10,9}	{10,10,9,8,10,6,10,10,8,10,10,10}	{Jan,Feb,Mar,Apr,May,Jun,Jul,Aug,Sep,Oct,Nov,Dec}
49326	48049	{7,7,7,8,9,9,9,10,9,10,10,10}	{10,9,9,8,10,10,10,10,10,9,9,10}	{Jan,Feb,Mar,Apr,May,Jun,Jul,Aug,Sep,Oct,Nov,Dec}
32410	16312	{6,7,7,7,9,8,9,9,9,10,9,9}	{7,10,9,8,10,8,8,10,8,10,8,9}	{Jan,Feb,Mar,Apr,May,Jun,Jul,Aug,Sep,Oct,Nov,Dec}
48880	16312	{10,10,10,8,9,9,9,10,9,7,6,6}	{9,10,9,8,10,10,7,10,8,5,8}	{Jan,Feb,Mar,Apr,May,Jun,Jul,Aug,Sep,Oct,Nov,Dec}
33288	16312	{10,8,9,10,9,9,9,10,9,8,10,10}	{10,10,9,8,10,7,10,10,10,10,8,10}	{Jan,Feb,Mar,Apr,May,Jun,Jul,Aug,Sep,Oct,Nov,Dec}
15795	16312	{6,7,8,8,10,9,9,10,9,8,10,9}	{10,5,9,8,10,9,10,10,8,10,10,10}	{Jan,Feb,Mar,Apr,May,Jun,Jul,Aug,Sep,Oct,Nov,Dec}
35761	16312	{4,7,7,8,9,10,9,10,9,10,8,10}	{9,10,6,8,10,10,8,10,10,10,9,10}	{Jan,Feb,Mar,Apr,May,Jun,Jul,Aug,Sep,Oct,Nov,Dec}
31276	16312	{5,7,7,5,6,5,9,8,9,9,10,9}	{5,6,6,7,6,8,10,10,10,10,10,10}	{Jan,Feb,Mar,Apr,May,Jun,Jul,Aug,Sep,Oct,Nov,Dec}
25285	36708	{6,7,7,8,9,9,9,10,9,10,10,10}	{10,10,9,8,10,10,10,10,10,10,10,10}	{Jan,Feb,Mar,Apr,May,Jun,Jul,Aug,Sep,Oct,Nov,Dec}
14903	36708	{10,7,8,8,9,9,9,10,9,10,9,10}	{10,10,9,8,10,8,10,9,10,10,10,10}	{Jan,Feb,Mar,Apr,May,Jun,Jul,Aug,Sep,Oct,Nov,Dec}
27489	36708	{7,7,7,6,9,10,9,7,9,10,9,10}	{10,10,9,8,10,9,10,5,10,10,10,10}	{Jan,Feb,Mar,Apr,May,Jun,Jul,Aug,Sep,Oct,Nov,Dec}
12236	36708	{10,7,7,8,9,8,8,10,9,9,10,8}	{10,10,9,8,10,10,10,10,10,7,7,8}	{Jan,Feb,Mar,Apr,May,Jun,Jul,Aug,Sep,Oct,Nov,Dec}
11284	36708	{4,7,7,9,9,9,5,10,9,10,9,10}	{2,10,9,8,10,10,8,10,8,10,10,10}	{Jan,Feb,Mar,Apr,May,Jun,Jul,Aug,Sep,Oct,Nov,Dec}

(20 rows)

First, we need to unnest the array data structure, which returns 12 distinct rows (in order) corresponding to the employee id that it belongs to. Then, using this unnested selection, we take an average of the employee's performance. This is carried out by grouping by employee id where the array order is equal to one through six, corresponding to months Jan - June. Now that we have an average performance rating for the first half for each employee, we do the same thing for the second half of the year. Next, we join these two nested queries using a with clause, which allows us to join the second half data on the first half data where the employee id's are the same. Once the join is complete, we filter based on first half performance being less than second half performance and get the below results.

case for employee paycut

employee_id	first_half_perf	second_half_perf
48880	9.3333333333333333	7.8333333333333333

(1 row)

case for employee raises

employee_id	first_half_perf	second_half_perf
16417	7.5000000000000000	9.0000000000000000
12236	8.1666666666666667	9.0000000000000000
25285	7.6666666666666667	9.6666666666666667
33288	9.1666666666666667	9.3333333333333333
45355	6.6666666666666667	8.8333333333333333
49326	7.8333333333333333	9.6666666666666667
12704	8.0000000000000000	9.0000000000000000
27489	7.6666666666666667	9.0000000000000000
38707	7.8333333333333333	9.3333333333333333
33542	7.5000000000000000	8.8333333333333333
31276	5.8333333333333333	9.0000000000000000
32410	7.3333333333333333	9.1666666666666667
14903	8.5000000000000000	9.5000000000000000
11284	7.5000000000000000	8.8333333333333333
35761	7.5000000000000000	9.3333333333333333
24141	7.1666666666666667	9.6666666666666667
15795	8.0000000000000000	9.1666666666666667
16096	7.6666666666666667	9.6666666666666667
10772	7.6666666666666667	8.1666666666666667

(19 rows)

Part 3: Triggers and functions

The goal of these triggers was to automate most of the constraint checks that we had to undertake on the front end of the project. For instance: ensuring that a primary key was not in the table, or that inputs would satisfy attribute constraints, and even allowing deletions from a table. The triggers all pertain to the menu_items table, which we allowed to be modified through the front end portion. In fact, that is how we tested the triggers by inserting and updating various fields.

First, we develop an audit table that tracks all of the modifications made to the underlying menu_items table. We deemed this feature important because we would like to know exactly what type of changes were made. This table is called menu_audit, and will contain tuples that are only inserted after modifications are made to the menu_items table.[1] The modification comes from the triggers that are set off before insertion, deletion, and updates.

menu_audit table

operation	timestamp	recipe_name	price	description
I	2021-04-02 23:45:05.861899	Toast	10.89	butter, maybe cream cheese?
I	2021-04-03 11:31:40.608728	toast2	16.33	N/A
I	2021-04-03 11:41:00.003942	huevos rancheros	14.70	fried eggs, corn tortilla, salsa fresca
I	2021-04-03 11:41:46.566238	overnight oats	9.25	
I	2021-04-03 11:42:26.038876	waffles with chorizo	18.51	
U	2021-04-03 11:43:13.92111	waffles with chorizo	18.51	sandwich with chorizo, eggs, and cheese
U	2021-04-03 11:44:00.07063	overnight oats	9.25	milk, oats, peanut butter and bananas
U	2021-04-03 11:45:06.038878	12-Hour Cold-Brewed Coffee	5.00	Coffee beans from Colombia
U	2021-04-03 11:45:41.88924	Plow Potatoes	10.62	
U	2021-04-03 11:46:31.532808	Plow Potatoes	10.62	home fries with sautéed veggies
U	2021-04-03 11:47:17.189541	Orange Juice	4.75	Pulp option and no pulp options
I	2021-04-03 11:53:10.135904	Slow-Cooker Mediterranean Frittata	19.05	eggs, cheese, veggies, olive oil,
D	2021-04-03 20:08:00.76278	test item	10.00	, Bread, Cheese,
D	2021-04-03 20:30:51.668991	toast2	16.33	N/A
D	2021-04-03 20:37:14.035322	Straus Whole Milk	3.50	
D	2021-04-03 20:38:03.088237	Mexican Coke	4.00	
U	2021-04-03 20:41:48.263576	Grapefruit Mimosa	14.88	
U	2021-04-03 20:42:24.359624	Grapefruit Mimosa	14.88	Grapefruit Juice, Chapagne
U	2021-04-03 20:42:58.034825	Iced Tea	4.30	
U	2021-04-03 20:43:33.263749	Iced Tea	4.30	Half Iced Tea Half Lemonade (with option to add alcohol :))
I	2021-04-03 20:52:58.51827	sheepherder's breakfast	17.30	Browned Bacon, fresh onion, hashed potatoes, 8 large eggs, gouda

(21 rows)

Trigger 1 & Function 1 -- $menu_insert$, menu_insert()

This function will return the menu_insert trigger, which is called before an insert into the menu_items table. The goal was, to do a more rigorous error checking, such that we could return more informative error messages when input was bad and ensure only good data is inserted. If you look at the sql code most of the error checks are very intuitive, except for the following. We declare a primary key counter variable, with which we select the count(*) of the recipe_name primary key that is to be inserted. If it is greater than zero, this is bad (can't have duplicate primary keys), else we finally get to the data manipulation. We thought that it would be interesting for customers to know how much the menu item would cost including sales tax.

[1] We adapted this audit table idea from the postgresql API, but we expanded on it by having 4 triggers not just the one. The main goal was to have an automated audit trail for any changes made to this core table for the restaurant business model. Source: https://www.postgresql.org/docs/12/plpgsql-trigger.html#PLPGSQL-TRIGGER-AUDIT-EXAMPLE

So we take the price being inputted and multiply it by an 8.875% sales tax, and return that tuple. However, before we return, we add a new row to the menu_audit table, saying that we are performing an Insert operation, and provide a timestamp. Finally, we chose to have the trigger pulled before an insert so we could modify the price to include the state sales tax.

Insert examples (success and failure)

```
project1=> insert into menu_items(recipe_name, price, description) values('Cold Brew', 5, null);
INSERT 0 1
```

```
project1=> insert into menu_items(recipe_name, price, description) values('Mimosa', 10, null);
ERROR:  Mimosa cant be a primary key it is already in the table
CONTEXT:  PL/pgSQL function menu_insert() line 26 at RAISE
```

Trigger 2 & Function 2 -- $menu_update_price$, menu_update_price()

This function will return the menu_update_price trigger, and will be called before an update to only a price in the menu. Since this code essentially does the same thing as in Trigger1, I will spare the details. We wanted to have a trigger that would keep track of when a user only updates the price for a given item. Furthermore, as in Trigger1, a tuple is entered into the menu_audit table corresponding to the update made to this row.

update price (success and failure)

```
project1=> update menu_items set price = 7 where recipe_name = 'Cold Brew';
UPDATE 1
```

```
project1=> update menu_items set price = -1 where recipe_name = 'Cold Brew';
ERROR:  Cold Brew cant be sold for negative dollars
CONTEXT:  PL/pgSQL function menu_update_price() line 6 at RAISE
```

Trigger 3 & Function 3 -- $menu_update_desc$, menu_update_desc()

This function will return the menu_update_desc trigger, and will be called before an update to only a description for a menu item. This code also does similar error checking to the text portion in Trigger1, but also inputs a tuple into menu_audit when the trigger is pulled.

update description (success)

```
project1=> update menu_items set description = 'freshly squeezed lemos' where recipe_name = 'House Made Lemonade';
UPDATE 1
```

Trigger 4 & Function 4 -- $menu_item_del$, menu_item_del()

This $menu_item_del$ trigger is one that we wanted to have from the beginning, since it makes sense to allow users to delete items from their menu table. However, this table has three foreign key references which makes any deletions very tricky. The trigger we wrote determines the eligibility for deletion and acts accordingly. Before we go into the details, we know that the trigger could have been implemented with a cascade, such that any foreign key references would automatically be deleted. But from a design standpoint, if you delete a menu_item, you would still want to have all of the order transactions that took place on this item. Cascading triggers would have deleted any reference which just didn't make sense to implement.

Thus, this trigger will query the three tables that are referenced and store a count(*) for this key in three different variables. Then, there are three if statements that check to see if the count(*) is greater than zero, and if it is, will raise an exception and rollback the transaction. If we make it through all of the three if statements and there are no references, then we can safely delete the menu_item and we also insert a tuple into menu_audit.

There are obvious limitations to this method, but we think that not cascading is better than removing all sales transactions on a particular key.

<u>delete tuple according to primary key (success and failure)</u>

```
delete from menu_items where recipe_name = 'Cold Brew';
DELETE 1
```

```
project1=> delete from menu_items where recipe_name = 'Mimosa';
ERROR:  cannot delete Mimosa, 1 foreign key reference(s) in food_sales table
CONTEXT:  PL/pgSQL function menu_item_del() line 14 at RAISE
```

These four triggers allow us to extend the functionality that we made in the part 3 front end, where a user has the ability to modify their restaurant menu. Through these triggers, we could better and more effectively check for bad inputs, make modifications to data points and store the new values, and of course keep track of all the changes made. Please see the attached **group31.sql** file ('Trigger code' section) for the code that we wrote to carry out these designs. Furthermore, to see the actual changes implemented with the code snippets, you can also look at the menu_audit table for more details.

Part 4 References:

1. https://www.postgresql.org/docs/12/plpgsql-trigger.html#PLPGSQL-TRIGGER-AUDIT-EXAMPLE
2. https://www.postgresql.org/docs/12/plpgsql-declarations.html
3. https://www.postgresql.org/docs/12/sql-createtrigger.html
4. https://stackoverflow.com/questions/12328198/store-query-result-in-a-variable-using-in-pl-pgsql
5. https://www.dbrnd.com/2016/11/postgresql-9-4-unnest-with-ordinality-to-generate-string-array-sequence-number/
6. https://hevodata.com/blog/postgresql-full-text-search-setup/