A FUNCTIONAL VIEW

Jed Rembold

Monday, November 25, 2024

ANNOUNCEMENTS

- Be making progress on your group project!
  - Presentations on Monday and Wednesday when we get back!
  - A list of learning objectives from HW over this semester will be posted tonight
- Grade reports: Should go out tonight I hope! All my attention is turning toward scoring HW8-9 now

PROJECT PRESENTATIONS

- Recall that you are presenting the week we get back!

Monday

- Jack & Connor
- Matthew & Aurora
- Sam H & Michael
- Greg & Marcus
- Grace & Haley
- Dayton & Nick

Wednesday

- Harleen & Sam J
- Jordan & Mallory
- AJ & Jerrick
- Hannah & Tippy
- Sergio & Evan
- Elise & Tiffany

DRY

- A common programming term is that of *DRY*: Don't repeat yourself
  - For the sake of convenience (why redo work you have already done?)
  - For the sake of debugging/maintenance (in case of a change, each replicate needs to be found and changed)
- Here we want to look at several methods in SQL that we can utilize to better adhere to DRY principles
  - Views
  - Custom functions
  - Triggers (on Tuesday)

ADMIRING THE VIEW

VIEWS

- A view in SQL is essentially a saved query output
- Existing views can be treated just like tables, selecting from their contents
  - You can even, to a limited extent, add, remove, or update contents
- Syntax similar to creating a new table, just with `VIEW`

```
CREATE VIEW view name AS
SELECT columns
FROM table
```

VIEW VS TABLE

- Despite seeming similar, views are fundamentally different from tables
  - A view stores no new internal information in the database, whereas creating a new table copies that information
    - This means a view will always reflect the latest data, whereas a created table would need to be updated
  - Views can give users access only to particular columns
    - This can be a boon for security and giving only those who need them permissions to change various tables

VIEW OPERATIONS

- You can drop views just like tables: `DROP VIEW` *view name*`;`
- You can create or *replace* views if the original columns and corresponding types are the same:
  - `CREATE OR REPLACE VIEW` *view name* `...`
  - Avoids having to drop first to recreate
  - New view could have more columns, but can not have less
- Insertions, updates and removes can be made on simple views, which will propagate back to the original table!
  - Views must reference single tables (no joins) and have no distinct or group by type clauses
  - Data in the original table not present in the view will be given `NULL` values (if not further restrictions on the view)

FURTHER LIMITING VIEWS

- Can also add `WITH LOCAL CHECK OPTION` to the end of a view creation query
- This will enforce that only data visible within the view can be edited with inserts, updates or removes
  - This includes any filtering done by the view!
- Can be an excellent way to limit what data can be changed
- Can replace `LOCAL` with `CASCADED` if you are referencing a view nested several times and want the restrictions to apply based on all the parent views as well
  - Needs to be applied at every nesting level

CUSTOM FUNCTIONS

FUNCTIONS

- Postgres has many prebuilt functions that we have used over the semester
- Sometimes you need something a bit more bespoke though, for which Postgres gives you some “easy” methods of writing your own functions
- What language you use to write your own function is highly flexible:
  - Pure SQL
  - PL/pgSQL: an extension of SQL that offers more programming logic
  - Python
- All functions will follow a similar syntax structure, but the body will depend on the details of your chosen language
  - Documentation for all the languages (and functions in general) can be found here

STRUCTURE OF FUNCTION

- The structure of any defined function will generally look like:

```
CREATE FUNCTION func_name(func_arguments)
RETURNS return_type AS
'function_body'
LANGUAGE func_lang;
```

- `func_name` - The desired name of the function
- `func_arguments` - A comma separated list of paired arguments and types
- `return_type` - The variable type of what will be returned (selected)
- `func_lang` - The language used for the function body
- `function_body` - a string containing the query or logic to process whenever the function is run

IN SQL

- `LANGUAGE SQL`
- Using the basic SQL language gives you access to any existing SQL queries that you already know
- To return something you should use the `SELECT`
- You can also write functions that do not return something by specifying `RETURNS void` and then using `INSERT`, `UPDATE`, `REMOVE`, etc.
- Using `$$` to quote the body is recommended so that you do not need to double up any single quotes

IN PGSQL

- `LANGUAGE plpgsql`
- Extends SQL to add classic programming control structures
- Needs to be wrapped in `BEGIN`, `END` keywords
- Operates like a hybrid between SQL and a more typical programming language
- Each phrase needs a terminating semicolon at the end

PGSQL STRUCTURES

- You can define variables

```
x := 56;
```

- Control statements

```
IF x > 5 THEN
   ...do something...
ELSE
   ...do something else...
END IF;
```

- Returns

```
RETURN x;
```

IN PYTHON

- `LANGUAGE plpython3u`
- Python is another supported language in which functions can be written
- Requires an extension to be active:

```
CREATE EXTENSION plpython3u;
```

- Most common data types with corresponding types are converted, anything else (like timestamps!) are just utilized as strings, so plan accordingly
- I’d generally write the Python in a more Python suited editor and then copy it in

ACTIVITY!

- I've given you a simple starter bit of sql here to create and populate some tables. Use them to practice the below in the shown pairings.
- Tasks:
  - Create a view called `newest` that will only show the "new" boxes from `best_boxes`
  - Create a view called `newred` that will only show new and red boxes using only `newest`
  - Adjust the `CHECK OPTION` setting for both views so that you could add `('Jane', 2, 7, 9, 'red', 'new')` to `newred` but not `('Bart', 6, 10, 12, 'red', 'used')` to `newred`
  - Write a function named `box_volume` to compute the volume of a cubic object, and use that function to populate a new `volume` column in `best_boxes`
  - Write a function called `scale_boxes` which updates the `best_boxes` table to scale the value in the named `width`, `height`, or `depth` columns by the provided argument. Run it to ensure it is working.

GROUP WORK!

- The remainder of class is set aside for you to work with your group on your projects!