# THE SCRIPTS INVOKED BY “0.SQL”

## cr-table.sql

Creates and populates a 12-row *(name, mgr_name)* table of emplyees. The ultimate manager has *mgr_name* set to *null.*

- Creates domain *name_t* with *language sql* constraint function *name_ok()* for business rule *“name must be lower case”*
- Creates table *emps(name name_t primary key, mgr_name name_t).*
- Inserts twelve rows.
- Creates constraint for business rule *“max one ultimate mgr”* (uses expression-based index).
- Creates constraint for business rule *“every emp except ultimate mgr has exacly one mgr”* (uses FK to same table).
- Selects table content ordered by *mgr_name nulls first, name.*
- Optionally tests the three business rules with three insert attempts.

## bare-recursive-cte.sql

Creates view *top_down_simple(depth, mgr_name, name).*

- Creates a view so that the logic it encapsulates can be re-used.
- Uses a recursive CTE to label each *emps* row with the *depth* in the hiearchy. The ultimate mgr is defined to have *“depth = 1”.*
- Displays view content ordered by *depth, mgr_name nulls first, name.*

## top-down-paths.sql

Creates view *top_down_paths(path)*

- Uses a recursive CTE to represent each *emps* row as the *array* of employees from the ultimate mgr down through successive reports to the current employee. The depth is implied by the number of elements (a.k.a. *cardinality*) of the array. The array is a convenient PostgreSQL construct to represent the implementation-agnostic notion of a *path.*
- Notice the following:

1) Using the array *constructor*, to get started by creating a single-element array (in the non-recursive term)
2) Using the *array concatenation operator* to append a new element to the array as it currently stands (in the recursive term)
3) Using the *cardinality()* function to identify the last element as *path[cardinality(path)].*

- Demonstrates breadth-first and then depth-first traversal.
- Conventional use of indentation to improve readability of the depth first traversal display.
- Approximation to Unix “tree” output by using └ and ├ and ──

## bottom-up-path.sql

Creates the prepared statement *bottom_up_simple(text)* parameterized by employee of interest.
Then reimplements the logic as the stored *language sql* function *bottom_up_path(start_name in text)* and wraps this with the stored *language plpgsql* function *bottom_up_path_display(start_name in text).*

- The logic encapsulated by *top_down_paths(path)* is inverted by having the non-recursive term establish the employee of interest.
- Then the recursive term searches *upwards* for the present employee’s manager.
- Using a *language sql* function is morally equivalent to using a prepared statement but is more suitable for real applications because you simply install it. (A prepared statement must be explicitly prepared afresh for each new session by client-side code).
- The *language plpgsql* wrapper function outputs a prettier display than the bare *::text* typecast of an array produces. Uses procedural code to iterate over the array’s elements to produce a nicely formatted *text* value. Trivial to do this procedurally. Tricky to do it declaratively.

# READING LIST

*blog posts*

Using the PostgreSQL Recursive CTE
**Part One: Traversing an employee hierarchy**
**Part Two: Computing Bacon Numbers for actors listed in the IMDb**

*YSQL documentation for SQL graph traversal*

**The WITH clause and common table expressions >**
**Case study—using a recursive CTE to traverse an employee hierarchy**

*Other YSQL with lots of code examples to run in PG and YB*

**JSON**
**arrays**
**window functions**
**aggregate functions**
**date-time**