Common Table Expressions (CTEs) allow queries to be more imperative, allowing looping and processing hierarchical structures that are normally associated only with imperative languages.

Creative Commons Attribution License http://momjian.us/presentations

Last updated: August, 2019
1. Imperative vs. declarative
2. Syntax
3. Recursive CTEs
4. Examples
5. Writable CTEs
6. Why use CTEs
1. Imperative vs. Declarative

https://www.flickr.com/photos/visit_cape_may/
In computer science, imperative programming is a programming paradigm that describes computation in terms of statements that change a program state. In much the same way that imperative mood in natural languages expresses commands to take action, imperative programs define sequences of commands for the computer to perform.

http://en.wikipedia.org/wiki/Imperative_programming
The term is used in opposition to declarative programming, which expresses what the program should accomplish without prescribing how to do it in terms of sequence.
Imperative

BASIC:

10 PRINT "Hello";
20 GOTO 10

C:

while (1)
    printf("Hello\n");

Perl:

print("Hello\n") while (1);
Declarative

SQL:

SELECT 'Hello'
UNION ALL
SELECT 'Hello'
UNION ALL
SELECT 'Hello'
UNION ALL
SELECT 'Hello'

...

An infinite loop is not easily implemented in simple SQL.
Imperative Database Options

- Client application code (e.g., libpq, JDBC, DBD::Pg)
- Server-side programming (e.g., PL/pgSQL, PL/Perl, C)
- Common table expressions
Common Table Expression (CTE) Syntax

WITH [ RECURSIVE ] with_query_name [( column_name [, ...] )] AS
  ( select ) [ , ... ]
SELECT ...
Keep Your Eye on the Red (Text)

https://www.flickr.com/photos/alltheaces/
WITH source AS (  
    SELECT 1  
  )  
SELECT * FROM source;  
?column?  
----------  
  1  

The CTE created a source table that was referenced by the outer SELECT. All queries in this presentation can be downloaded from http://momjian.us/main/writings/pgsql/cte.sql.
WITH source AS (  
    SELECT 1 AS col1
 )
SELECT * FROM source;

<table>
<thead>
<tr>
<th>col1</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
</tr>
</tbody>
</table>

The CTE returned column is `source.col1`. 
WITH source (col1) AS (  
    SELECT 1
  )
SELECT * FROM source;
  col1
-----
   1
WITH source (col2) AS (  
    SELECT 1 AS col1  
  )  
SELECT col2 AS col3 FROM source;  
col3  
-----  
    1  
The CTE column starts as *col1*, is renamed in the WITH clause as *col2*, and the outer SELECT renames it to *col3*.  

Columns Can Be Renamed
Multiple CTE Columns Can Be Returned

WITH source AS (  
    SELECT 1, 2  
)  
SELECT * FROM source;  
?column? | ?column?  
---------+---------  
  1       |  2
SELECT 1
UNION
SELECT 1;
<table>
<thead>
<tr>
<th>?column?</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
</tr>
</tbody>
</table>

SELECT 1
UNION ALL
SELECT 1;
<table>
<thead>
<tr>
<th>?column?</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
</tr>
<tr>
<td>1</td>
</tr>
</tbody>
</table>
WITH source AS (  
    SELECT 1, 2  
  ),

  source2 AS (  
    SELECT 3, 4  
  )

SELECT * FROM source
UNION ALL
SELECT * FROM source2;

<table>
<thead>
<tr>
<th>column?</th>
<th>column?</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
</tr>
</tbody>
</table>
WITH source AS (  
    SELECT lanname, rolname  
    FROM pg_language JOIN pg_roles ON lanowner = pg_roles.oid  
)  
SELECT * FROM source;  
<table>
<thead>
<tr>
<th>lanname</th>
<th>rolname</th>
</tr>
</thead>
<tbody>
<tr>
<td>internal</td>
<td>postgres</td>
</tr>
<tr>
<td>c</td>
<td>postgres</td>
</tr>
<tr>
<td>sql</td>
<td>postgres</td>
</tr>
<tr>
<td>plpgsql</td>
<td>postgres</td>
</tr>
</tbody>
</table>
WITH source AS (  
    SELECT lanname, rolname  
    FROM pg_language JOIN pg_roles ON lanowner = pg_roles.oid  
    ORDER BY lanname
  )
SELECT * FROM source  
UNION ALL  
SELECT MIN(lanname), NULL  
FROM source;

<table>
<thead>
<tr>
<th>lanname</th>
<th>rolname</th>
</tr>
</thead>
<tbody>
<tr>
<td>c</td>
<td>postgres</td>
</tr>
<tr>
<td>internal</td>
<td>postgres</td>
</tr>
<tr>
<td>plpgsql</td>
<td>postgres</td>
</tr>
<tr>
<td>sql</td>
<td>postgres</td>
</tr>
<tr>
<td>c</td>
<td></td>
</tr>
</tbody>
</table>
WITH class AS ( 
    SELECT oid, relname 
    FROM pg_class 
    WHERE relkind = 'r' 
)

SELECT class.relname, attname 
FROM pg_attribute, class 
WHERE class.oid = attrelid 
ORDER BY 1, 2 
LIMIT 5;

<table>
<thead>
<tr>
<th>relname</th>
<th>attname</th>
</tr>
</thead>
<tbody>
<tr>
<td>pg_aggregate</td>
<td>aggfinalfn</td>
</tr>
<tr>
<td>pg_aggregate</td>
<td>aggfnooid</td>
</tr>
<tr>
<td>pg_aggregate</td>
<td>agginitval</td>
</tr>
<tr>
<td>pg_aggregate</td>
<td>aggsortop</td>
</tr>
<tr>
<td>pg_aggregate</td>
<td>aggtransfn</td>
</tr>
</tbody>
</table>
CASE
WHEN condition THEN result
ELSE result
END

For example:
SELECT col,
CASE
  WHEN col > 0 THEN 'positive'
  WHEN col = 0 THEN 'zero'
  ELSE 'negative'
END
FROM tab;
3. Recursive CTEs

https://www.flickr.com/photos/rbh/
WITH RECURSIVE source AS ( 
    SELECT 1  
)
SELECT * FROM source;

This does not loop because source is not mentioned in the CTE.
This Is an Infinite Loop

```
SET statement_timeout = '1s';

WITH RECURSIVE source AS (  
    SELECT 1  
    UNION ALL  
    SELECT 1 FROM source  
)
SELECT * FROM source;
ERROR: canceling statement due to statement timeout
```
WITH RECURSIVE source AS ( 
SELECT 1
UNION ALL
SELECT 1 FROM source
)
SELECT * FROM source;
WITH RECURSIVE source AS (  
    SELECT 'Hello'  
    UNION ALL  
    SELECT 'Hello' FROM source  
)  
SELECT * FROM source;  
ERROR: canceling statement due to statement timeout

RESET statement_timeout;
WITH RECURSIVE source AS ( 
    SELECT 'Hello' 
    UNION 
    SELECT 'Hello' FROM source 
) 
SELECT * FROM source; 

Hello
WITH RECURSIVE source (counter) AS (  
    -- seed value
    SELECT 1  
    UNION ALL  
    SELECT counter + 1  
    FROM source  
    -- terminal condition  
    WHERE counter < 10
)
SELECT * FROM source;
Of course, this can be more easily accomplished using `generate_series(1, 10)`.
Perl Example

```perl
for (my $i = 1; $i <= 10; $i++)
{
    print "$i\n";
}
```
sub f
{
    my $arg = shift;
    print "$arg\n";
    f($arg + 1) if ($arg < 10);
}

f(1);
my @table;
sub f
{
    my $arg = shift // 1;
    push @table, $arg;
    f($arg + 1) if ($arg < 10);
}
f();
map {print "$_ \n"} @table;

This is the most accurate representation of CTEs because it accumulates results in an array (similar to a table result).
4. Examples

https://www.flickr.com/photos/82134796@N03/
WITH RECURSIVE source (counter, product) AS (  
    SELECT 1, 1  
    UNION ALL  
    SELECT counter + 1, product * (counter + 1)  
    FROM source  
    WHERE counter < 10  
)
SELECT counter, product FROM source;
| counter | product |
|---------+---------|
| 1       | 1       |
| 2       | 2       |
| 3       | 6       |
| 4       | 24      |
| 5       | 120     |
| 6       | 720     |
| 7       | 5040    |
| 8       | 40320   |
| 9       | 362880  |
| 10      | 3628800 |
WITH RECURSIVE source (counter, product) AS (  
    SELECT 1, 1  
    UNION ALL  
    SELECT counter + 1, product * (counter + 1)  
    FROM source  
    WHERE counter < 10  
)
SELECT counter, product  
FROM source  
WHERE counter = 10;

<table>
<thead>
<tr>
<th>counter</th>
<th>product</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>3628800</td>
</tr>
</tbody>
</table>
my @table;
sub f {
  my ($counter, $product) = @_; 
  my ($counter_new, $product_new);
  if (!defined($counter)) {
    $counter_new = 1;
    $product_new = 1;
  } else {
    $counter_new = $counter + 1;
    $product_new = $product * ($counter + 1);
  }
  push(@table, [$counter_new, $product_new]);
  f($counter_new, $product_new) if ($counter < 10);
}
f();
map {print @$_ \n if ($_->[0]) == 10} @table;
String Manipulation Is Also Possible

WITH RECURSIVE source (str) AS (  
  SELECT 'a' 
  UNION ALL 
  SELECT str || 'a' 
  FROM source 
  WHERE length(str) < 10 
)
SELECT * FROM source;
str
        ---------
a
aa
aaa
aaaa
aaaaa
aaaaaa
aaaaaaa
aaaaaaaa
aaaaaaaaa
aaaaaaaaaa
aaaaaaaaaaa
aaaaaaaaaaaa
aaaaaaaaaaaaa
aaaaaaaaaaaaaa
WITH RECURSIVE source (str) AS ( 
    SELECT 'a'
    UNION ALL
    SELECT str || chr(ascii(substr(str, length(str))) + 1)
    FROM source
    WHERE length(str) < 10
)
SELECT * FROM source;
Output

str
---------
a
ab
abc
abcd
abcde
abcdef
abcdefg
abcdefgh
abcdefghi
abcdefghij
WITH RECURSIVE source (counter) AS (  
    SELECT -10  
    UNION ALL  
    SELECT counter + 1  
    FROM source  
    WHERE counter < 10  
  )  
SELECT repeat(' ', 5 - abs(counter) / 2) || 'X' || repeat(' ', abs(counter)) || 'X'  
FROM source;
WITH RECURSIVE source (counter) AS (  
    SELECT -10  
    UNION ALL  
    SELECT counter + 1  
    FROM source  
    WHERE counter < 10  
  )  
SELECT counter,  
repeat(' ', 5 - abs(counter) / 2) ||  
'X' ||  
repeat(' ', abs(counter)) ||  
'X'  
FROM source;

This generates Integers from -10 to 10, and these numbers are used to print an appropriate number of spaces.
<table>
<thead>
<tr>
<th>counter</th>
<th>column?</th>
</tr>
</thead>
<tbody>
<tr>
<td>-10</td>
<td>X X</td>
</tr>
<tr>
<td>-9</td>
<td>X X</td>
</tr>
<tr>
<td>-8</td>
<td>X X</td>
</tr>
<tr>
<td>-7</td>
<td>X X</td>
</tr>
<tr>
<td>-6</td>
<td>X X</td>
</tr>
<tr>
<td>-5</td>
<td>X X</td>
</tr>
<tr>
<td>-4</td>
<td>X X</td>
</tr>
<tr>
<td>-3</td>
<td>X X</td>
</tr>
<tr>
<td>-2</td>
<td>X X</td>
</tr>
<tr>
<td>-1</td>
<td>X X</td>
</tr>
<tr>
<td>0</td>
<td>XX</td>
</tr>
<tr>
<td>1</td>
<td>X X</td>
</tr>
<tr>
<td>2</td>
<td>X X</td>
</tr>
<tr>
<td>3</td>
<td>X X</td>
</tr>
<tr>
<td>4</td>
<td>X X</td>
</tr>
<tr>
<td>5</td>
<td>X X</td>
</tr>
<tr>
<td>6</td>
<td>X X</td>
</tr>
<tr>
<td>7</td>
<td>X X</td>
</tr>
<tr>
<td>8</td>
<td>X X</td>
</tr>
<tr>
<td>9</td>
<td>X X</td>
</tr>
<tr>
<td>10</td>
<td>X X</td>
</tr>
</tbody>
</table>
WITH RECURSIVE source (counter) AS ( 
    SELECT -10
    UNION ALL
    SELECT counter + 1
    FROM source
    WHERE counter < 10
) 
SELECT repeat(' ', abs(counter)/2) || 'X' || repeat(' ', 10 - abs(counter)) || 'X'
FROM source;
WITH RECURSIVE source (counter) AS (  
    SELECT -10  
    UNION ALL  
    SELECT counter + 1  
    FROM source  
    WHERE counter < 10  
)
SELECT repeat(' ', int4(pow(counter, 2)/10)) || 'X' || repeat(' ', 2 * (10 - int4(pow(counter, 2)/10))) || 'X'
FROM source;
An Oval

?column?

---------------------------------
XX
X  X
X  X
X  X
X  X
X  X
X  X
X  X
X  X
X  X
X  X
X  X
X  X
X  X
X  X
X  X
X  X
X  X
X  X
X  X
X  X
XX
WITH RECURSIVE source (counter) AS (  
    SELECT -10  
    UNION ALL  
    SELECT counter + 1  
    FROM source  
    WHERE counter < 10  
  )
SELECT repeat(' ', int4(pow(counter, 2)/5)) ||  
    'X' ||  
repeat(' ', 2 * (20 - int4(pow(counter, 2)/5))) ||  
'X'  
FROM source;
The prime factors of $X$ are the prime numbers that must be multiplied to equal a $X$, e.g.:

$10 = 2 \times 5$
$27 = 3 \times 3 \times 3$
$48 = 2 \times 2 \times 2 \times 2 \times 3$
$66 = 2 \times 3 \times 11$
$70 = 2 \times 5 \times 7$
$100 = 2 \times 2 \times 5 \times 5$
WITH RECURSIVE source (counter, factor, is_factor) AS (  
    SELECT 2, 56, false  
    UNION ALL  
    SELECT  
        CASE WHEN factor % counter = 0 THEN counter  
            ELSE counter + 1  
        END,  
        CASE WHEN factor % counter = 0 THEN factor / counter  
            ELSE factor  
        END,  
        CASE WHEN factor % counter = 0 THEN true  
            ELSE false  
        END  
    FROM source  
    WHERE factor <> 1  
)  
SELECT * FROM source;
<table>
<thead>
<tr>
<th>counter</th>
<th>factor</th>
<th>is_factor</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>56</td>
<td>f</td>
</tr>
<tr>
<td>2</td>
<td>28</td>
<td>t</td>
</tr>
<tr>
<td>2</td>
<td>14</td>
<td>t</td>
</tr>
<tr>
<td>2</td>
<td>7</td>
<td>t</td>
</tr>
<tr>
<td>3</td>
<td>7</td>
<td>f</td>
</tr>
<tr>
<td>4</td>
<td>7</td>
<td>f</td>
</tr>
<tr>
<td>5</td>
<td>7</td>
<td>f</td>
</tr>
<tr>
<td>6</td>
<td>7</td>
<td>f</td>
</tr>
<tr>
<td>7</td>
<td>7</td>
<td>f</td>
</tr>
<tr>
<td>7</td>
<td>1</td>
<td>t</td>
</tr>
</tbody>
</table>
WITH RECURSIVE source (counter, factor, is_factor) AS ( 
    SELECT 2, 56, false 
    UNION ALL 
    SELECT 
        CASE 
        WHEN factor % counter = 0 THEN counter 
        ELSE counter + 1 
        END, 
        CASE 
        WHEN factor % counter = 0 THEN factor / counter 
        ELSE factor 
        END, 
        CASE 
        WHEN factor % counter = 0 THEN true 
        ELSE false 
        END 
    FROM source 
    WHERE factor <> 1 
) 
SELECT * FROM source WHERE is_factor;
<table>
<thead>
<tr>
<th>counter</th>
<th>factor</th>
<th>is_factor</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>28</td>
<td>t</td>
</tr>
<tr>
<td>2</td>
<td>14</td>
<td>t</td>
</tr>
<tr>
<td>2</td>
<td>7</td>
<td>t</td>
</tr>
<tr>
<td>2</td>
<td>7</td>
<td>t</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>t</td>
</tr>
<tr>
<td>7</td>
<td>1</td>
<td>t</td>
</tr>
</tbody>
</table>
WITH RECURSIVE source (counter, factor, is_factor) AS (  
    SELECT 2, 322434, false  
    UNION ALL  
    SELECT  
        CASE  
            WHEN factor % counter = 0 THEN counter  
        ELSE counter + 1  
        END,  
        CASE  
            WHEN factor % counter = 0 THEN factor / counter  
        ELSE factor  
        END,  
        CASE  
            WHEN factor % counter = 0 THEN true  
        ELSE false  
        END  
    FROM source  
    WHERE factor <> 1  
)  
SELECT * FROM source WHERE is_factor;
<table>
<thead>
<tr>
<th>counter</th>
<th>factor</th>
<th>is_factor</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>161217</td>
<td>t</td>
</tr>
<tr>
<td>3</td>
<td>53739</td>
<td>t</td>
</tr>
<tr>
<td>3</td>
<td>17913</td>
<td>t</td>
</tr>
<tr>
<td>3</td>
<td>5971</td>
<td>t</td>
</tr>
<tr>
<td>7</td>
<td>853</td>
<td>t</td>
</tr>
<tr>
<td>853</td>
<td>1</td>
<td>t</td>
</tr>
</tbody>
</table>
WITH RECURSIVE source (counter, factor, is_factor) AS (  
    SELECT 2, 66, false  
    UNION ALL  
    SELECT  
        CASE  
            WHEN factor % counter = 0 THEN counter  
            ELSE counter + 1  
        END,  
        CASE  
            WHEN factor % counter = 0 THEN factor / counter  
            ELSE factor  
        END,  
        CASE  
            WHEN factor % counter = 0 THEN true  
            ELSE false  
        END  
    FROM source  
WHERE factor <> 1)
SELECT * FROM source;
<table>
<thead>
<tr>
<th>counter</th>
<th>factor</th>
<th>is_factor</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>66</td>
<td>f</td>
</tr>
<tr>
<td>2</td>
<td>33</td>
<td>t</td>
</tr>
<tr>
<td>3</td>
<td>33</td>
<td>f</td>
</tr>
<tr>
<td>3</td>
<td>11</td>
<td>t</td>
</tr>
<tr>
<td>4</td>
<td>11</td>
<td>f</td>
</tr>
<tr>
<td>5</td>
<td>11</td>
<td>f</td>
</tr>
<tr>
<td>6</td>
<td>11</td>
<td>f</td>
</tr>
<tr>
<td>7</td>
<td>11</td>
<td>f</td>
</tr>
<tr>
<td>8</td>
<td>11</td>
<td>f</td>
</tr>
<tr>
<td>9</td>
<td>11</td>
<td>f</td>
</tr>
<tr>
<td>10</td>
<td>11</td>
<td>f</td>
</tr>
<tr>
<td>11</td>
<td>11</td>
<td>f</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>t</td>
</tr>
</tbody>
</table>

Inefficient
WITH RECURSIVE source (counter, factor, is_factor) AS (  
    SELECT 2, 66, false  
    UNION ALL  
    SELECT  
        CASE  
            WHEN factor % counter = 0 THEN counter  
            -- is 'factor' prime?  
            WHEN counter * counter > factor THEN factor  
            -- now only odd numbers  
            WHEN counter = 2 THEN 3  
            ELSE counter + 2  
        END,  
        CASE  
            WHEN factor % counter = 0 THEN factor / counter  
            ELSE factor  
        END,  
        CASE  
            WHEN factor % counter = 0 THEN true  
            ELSE false  
        END  
    FROM source  
    WHERE factor <> 1  
)  
SELECT * FROM source;
<table>
<thead>
<tr>
<th>counter</th>
<th>factor</th>
<th>is_factor</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>66</td>
<td>f</td>
</tr>
<tr>
<td>2</td>
<td>33</td>
<td>t</td>
</tr>
<tr>
<td>3</td>
<td>33</td>
<td>f</td>
</tr>
<tr>
<td>3</td>
<td>11</td>
<td>t</td>
</tr>
<tr>
<td>5</td>
<td>11</td>
<td>f</td>
</tr>
<tr>
<td>11</td>
<td>11</td>
<td>f</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>t</td>
</tr>
</tbody>
</table>
WITH RECURSIVE source (counter, factor, is_factor) AS (  
    SELECT 2, 66, false  
    UNION ALL  
    SELECT  
        CASE  
            WHEN factor % counter = 0 THEN counter  
            -- is 'factor' prime?  
            WHEN counter * counter > factor THEN factor  
            -- now only odd numbers  
            WHEN counter = 2 THEN 3  
            ELSE counter + 2  
        END,  
        CASE  
            WHEN factor % counter = 0 THEN factor / counter  
            ELSE factor  
        END,  
        CASE  
            WHEN factor % counter = 0 THEN true  
            ELSE false  
        END  
    FROM source  
    WHERE factor <> 1  
)  
SELECT * FROM source WHERE is_factor;
<table>
<thead>
<tr>
<th>counter</th>
<th>factor</th>
<th>is_factor</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>33</td>
<td>t</td>
</tr>
<tr>
<td>3</td>
<td>11</td>
<td>t</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>t</td>
</tr>
</tbody>
</table>
my @table;
sub f
{
    my ($counter, $factor, $is_factor) = @_;
    my ($counter_new, $factor_new, $is_factor_new);
    if (!defined($counter)) {
        $counter_new = 2;
        $factor_new = 66;
        $is_factor_new = 0;
    } else {
        $counter_new = ($factor % $counter == 0) ? $counter :
            ($counter * $counter > $factor) ? $factor :
            ($counter == 2) ? 3 :
            $counter + 2;
        $factor_new = ($factor % $counter == 0) ? $factor / $counter :
            $factor;
        $is_factor_new = ($factor % $counter == 0);
    }
    push(@table, [$counter_new, $factor_new, $is_factor_new]);
    f($counter_new, $factor_new) if ($factor != 1);
}
f();
map {print "$_->[0] $_->[1] $_->[2]\n" if ($_->[2]) == 1} @table;
CREATE TEMPORARY TABLE part (parent_part_no INTEGER, part_no INTEGER);

INSERT INTO part VALUES (1, 11);
INSERT INTO part VALUES (1, 12);
INSERT INTO part VALUES (1, 13);
INSERT INTO part VALUES (2, 21);
INSERT INTO part VALUES (2, 22);
INSERT INTO part VALUES (2, 23);
INSERT INTO part VALUES (11, 101);
INSERT INTO part VALUES (13, 102);
INSERT INTO part VALUES (13, 103);
INSERT INTO part VALUES (22, 221);
INSERT INTO part VALUES (22, 222);
INSERT INTO part VALUES (23, 231);
WITH RECURSIVE source (part_no) AS (  
    SELECT 2  
    UNION ALL  
    SELECT part.part_no  
        FROM source JOIN part ON (source.part_no = part.parent_part_no)  
)  
SELECT * FROM source;

```
part_no
---------
    2
   21
   22
   23
  221
  222
  231
```

Using UNION without ALL here would avoid infinite recursion if there is a loop in the data, but it would also cause a part with multiple parents to appear only once.
WITH RECURSIVE source (level, part_no) AS ( 
    SELECT 0, 2 
    UNION ALL 
    SELECT level + 1, part.part_no 
    FROM source JOIN part ON (source.part_no = part.parent_part_no) 
)

SELECT '+' || repeat('-', level * 2) || part_no::text AS part_tree FROM source;

---

+2
+-21
+-22
+-23
+--221
+--222
+--231
WITH RECURSIVE source (level, tree, part_no) AS ( 
    SELECT 0, '2', 2 
    UNION ALL 
    SELECT level + 1, tree || ' ' || part.part_no::text, part.part_no 
    FROM source JOIN part ON (source.part_no = part.parent_part_no) 
)

SELECT '+' || repeat('-', level * 2) || part_no::text AS part_tree, tree 
FROM source 
ORDER BY tree;

<table>
<thead>
<tr>
<th>part_tree</th>
<th>tree</th>
</tr>
</thead>
<tbody>
<tr>
<td>+2</td>
<td>2</td>
</tr>
<tr>
<td>+--21</td>
<td>2 21</td>
</tr>
<tr>
<td>+--22</td>
<td>2 22</td>
</tr>
<tr>
<td>+----221</td>
<td>2 22 221</td>
</tr>
<tr>
<td>+----222</td>
<td>2 22 222</td>
</tr>
<tr>
<td>+--23</td>
<td>2 23</td>
</tr>
<tr>
<td>+-----231</td>
<td>2 23 231</td>
</tr>
</tbody>
</table>
WITH RECURSIVE source (level, tree, part_no) AS (
    SELECT 0, '{2}::int[]', 2
    UNION ALL
    SELECT level + 1, array_append(tree, part.part_no), part.part_no
    FROM source JOIN part ON (source.part_no = part.parent_part_no)
)
SELECT '+' || repeat('-', level * 2) || part_no::text AS part_tree, tree
FROM source
ORDER BY tree;

<table>
<thead>
<tr>
<th>part_tree</th>
<th>tree</th>
</tr>
</thead>
<tbody>
<tr>
<td>+2</td>
<td>{2}</td>
</tr>
<tr>
<td>+--21</td>
<td>{2,21}</td>
</tr>
<tr>
<td>+--22</td>
<td>{2,22}</td>
</tr>
<tr>
<td>+----221</td>
<td>{2,22,221}</td>
</tr>
<tr>
<td>+----222</td>
<td>{2,22,222}</td>
</tr>
<tr>
<td>+--23</td>
<td>{2,23}</td>
</tr>
<tr>
<td>+----231</td>
<td>{2,23,231}</td>
</tr>
</tbody>
</table>
WITH RECURSIVE source (level, tree, part_no) AS ( 
    SELECT 0, '{2}'::int[], 2 
    UNION ALL 
    SELECT level + 1, array_append(tree, part.part_no), part.part_no 
    FROM source JOIN part ON (source.part_no = part.parent_part_no) 
) 

SELECT *, '+' || repeat('-', level * 2) || part_no::text AS part_tree 
FROM source 
ORDER BY tree; 

| level | tree  | part_no | part tree  |
|-------+-------+---------+------------|
| 0     | {2}   | 2       | +2         |
| 1     | {2,21}| 21      | ++21       |
| 1     | {2,22}| 22      | ++22       |
| 2     | {2,22,221} | 221 | +++221     |
| 2     | {2,22,222} | 222 | +++222     |
| 1     | {2,23}| 23      | ++23       |
| 2     | {2,23,231} | 231 | +++231     |
CREATE TEMPORARY TABLE deptest (x1 INTEGER);
WITH RECURSIVE dep (classid, obj) AS ( 
    SELECT (SELECT oid FROM pg_class WHERE relname = 'pg_class'),
           oid
    FROM pg_class
    WHERE relname = 'deptest'
    UNION ALL
    SELECT pg_depend.classid, objid
    FROM pg_depend JOIN dep ON (refobjid = dep.obj)
  )
SELECT (SELECT relname FROM pg_class WHERE oid = classid) AS class,
        (SELECT typname FROM pg_type WHERE oid = obj) AS type,
        (SELECT relname FROM pg_class WHERE oid = obj) AS class,
        (SELECT relkind FROM pg_class where oid = obj::regclass) AS kind,
        (SELECT conname FROM pg_constraint WHERE oid = obj) AS constraint
FROM dep
ORDER BY obj;
<table>
<thead>
<tr>
<th>class</th>
<th>type</th>
<th>class</th>
<th>kind</th>
<th>attrdef</th>
<th>constraint</th>
</tr>
</thead>
<tbody>
<tr>
<td>pg_class</td>
<td></td>
<td>deptest</td>
<td>r</td>
<td></td>
<td></td>
</tr>
<tr>
<td>pg_type</td>
<td>_deptest</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>pg_type</td>
<td>deptest</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
WITH RECURSIVE dep (classid, obj) AS (
    SELECT classid, objid
    FROM pg_depend JOIN pg_class ON (refobjid = pg_class.oid)
    WHERE relname = 'deptest'
    UNION ALL
    SELECT pg_depend.classid, objid
    FROM pg_depend JOIN dep ON (refobjid = dep.obj)
)
SELECT (SELECT relname FROM pg_class WHERE oid = classid) AS class,
       (SELECT typname FROM pg_type WHERE oid = obj) AS type,
       (SELECT relname FROM pg_class WHERE oid = obj) AS class,
       (SELECT relkind FROM pg_class where oid = obj::regclass) AS kind,
       (SELECT adsrc FROM pg_attrdef WHERE oid = obj) AS attrdef,
       (SELECT conname FROM pg_constraint WHERE oid = obj) AS constraint
FROM dep
ORDER BY obj;
<table>
<thead>
<tr>
<th>class</th>
<th>type</th>
<th>class</th>
<th>kind</th>
<th>attrdef</th>
<th>constraint</th>
</tr>
</thead>
<tbody>
<tr>
<td>pg_type</td>
<td>_deptest</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>pg_type</td>
<td>deptest</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
ALTER TABLE deptest ADD PRIMARY KEY (x1);

NOTICE: ALTER TABLE / ADD PRIMARY KEY will create implicit index "deptest_pkey" for table "deptest"
WITH RECURSIVE dep (classid, obj) AS (  
    SELECT (SELECT oid FROM pg_class WHERE relname = 'pg_class'),
    oid
    FROM pg_class
    WHERE relname = 'deptest'
    UNION ALL
    SELECT pg_depend.classid, objid
    FROM pg_depend JOIN dep ON (refobjid = dep.obj)
)
SELECT (SELECT relname FROM pg_class WHERE oid = classid) AS class,
    (SELECT typname FROM pg_type WHERE oid = obj) AS type,
    (SELECT relname FROM pg_class WHERE oid = obj) AS class,
    (SELECT relkind FROM pg_class where oid = obj::regclass) AS kind,
    (SELECT adsrc FROM pg_attrdef WHERE oid = obj) AS attrdef,
    (SELECT conname FROM pg_constraint WHERE oid = obj) AS constraint
FROM dep
ORDER BY obj;
<table>
<thead>
<tr>
<th>class</th>
<th>type</th>
<th>class</th>
<th>kind</th>
<th>attrdef</th>
<th>constraint</th>
</tr>
</thead>
<tbody>
<tr>
<td>pg_class</td>
<td></td>
<td>deptest</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>pg_type</td>
<td>_deptest</td>
<td></td>
<td></td>
<td></td>
<td>deptest</td>
</tr>
<tr>
<td>pg_type</td>
<td>deptest</td>
<td></td>
<td></td>
<td></td>
<td>deptest</td>
</tr>
<tr>
<td>pg_class</td>
<td></td>
<td>deptest_pkey</td>
<td>i</td>
<td></td>
<td>deptest_pkey</td>
</tr>
<tr>
<td>pg_constraint</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>deptest_pkey</td>
</tr>
</tbody>
</table>
ALTER TABLE deptest ADD COLUMN x2 SERIAL;
NOTICE: ALTER TABLE will create implicit sequence "deptest_x2_seq" for serial column "deptest.x2"
WITH RECURSIVE dep (classid, obj) AS (  
SELECT (SELECT oid FROM pg_class WHERE relname = 'pg_class'),  
      oid  
FROM pg_class  
WHERE relname = 'deptest'  
UNION ALL  
SELECT pg_depend.classid, objid  
FROM pg_depend JOIN dep ON (refobjid = dep.obj)  
)
SELECT (SELECT relname FROM pg_class WHERE oid = classid) AS class,  
       (SELECT typname FROM pg_type WHERE oid = obj) AS type,  
       (SELECT relname FROM pg_class WHERE oid = obj) AS class,  
       (SELECT relkind FROM pg_class where oid = obj::regclass) AS kind,  
       (SELECT adsrc FROM pg_attrdef WHERE oid = obj) AS attrdef  
-- column removed to reduce output width  
FROM dep  
ORDER BY obj;
| class        | type  | class    | kind | attrdef                                                   |
|--------------|-------|----------|------|---------------|--------------------------------------------------------|
| pg_class     |       | deptest  | r    |               |
| pg_type      | _deptest | | | |
| pg_type      | deptest | | | |
| pg_class     |       | deptest_pkey | i    |               |
| pg_constraint|       |          |      |               |
| pg_class     |       | deptest_x2_seq | S    |               |
| pg_type      | deptest_x2_seq | | | |
| pg_attrdef   |       |          |      | nextval('deptest_x2_seq'::regclass) |
| pg_attrdef   |       |          |      | nextval('deptest_x2_seq'::regclass) |
WITH RECURSIVE dep (level, tree, classid, obj) AS (
    SELECT 0, array_append(null, oid)::oid[],
           (SELECT oid FROM pg_class WHERE relname = 'pg_class'),
           oid
    FROM pg_class
    WHERE relname = 'deptest'
    UNION ALL
    SELECT level + 1, array_append(tree, objid),
           pg_depend.classid, objid
    FROM pg_depend JOIN dep ON (refobjid = dep.obj)
)
SELECT tree,
       (SELECT relname FROM pg_class WHERE oid = classid) AS class,
       (SELECT typname FROM pg_type WHERE oid = obj) AS type,
       (SELECT relname FROM pg_class WHERE oid = obj) AS class,
       (SELECT relkind FROM pg_class where oid = obj::regclass) AS kind
       -- column removed to reduce output width
FROM dep
ORDER BY tree, obj;
<table>
<thead>
<tr>
<th>tree</th>
<th>class</th>
<th>type</th>
<th>class</th>
<th>kind</th>
</tr>
</thead>
<tbody>
<tr>
<td>{16458}</td>
<td>pg_class</td>
<td></td>
<td>deptest</td>
<td>r</td>
</tr>
<tr>
<td>{16458,16460}</td>
<td>pg_type</td>
<td>deptest</td>
<td></td>
<td></td>
</tr>
<tr>
<td>{16458,16460,16459}</td>
<td>pg_type</td>
<td>_deptest</td>
<td></td>
<td></td>
</tr>
<tr>
<td>{16458,16462}</td>
<td>pg_constraint</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>{16458,16462,16461}</td>
<td>pg_class</td>
<td></td>
<td>deptest_pkey</td>
<td>i</td>
</tr>
<tr>
<td>{16458,16463}</td>
<td>pg_class</td>
<td></td>
<td>deptest_x2_seq</td>
<td>S</td>
</tr>
<tr>
<td>{16458,16463,16464}</td>
<td>pg_type</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>{16458,16463,16465}</td>
<td>pg_attrdef</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>{16458,16465}</td>
<td>pg_attrdef</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
5. Writable CTEs

https://www.flickr.com/photos/dmelchordiaz/
Writable CTEs

- Allow data-modification commands (INSERT/UPDATE/DELETE) in WITH clauses
  - These commands can use RETURNING to pass data up to the containing query.
- Allow WITH clauses to be attached to INSERT, UPDATE, DELETE statements
CREATE TEMPORARY TABLE retdemo (x NUMERIC);

INSERT INTO retdemo VALUES (random()), (random()), (random()) RETURNING x;

----------------------
0.00761545216664672
0.85416117589920831
0.10137318633496895

WITH source AS (  
    INSERT INTO retdemo  
    VALUES (random()), (random()), (random()) RETURNING x
)
SELECT AVG(x) FROM source;
  avg
----------------------
0.46403147140517833
WITH source AS ( 
    DELETE FROM retdemo RETURNING x 
)
SELECT MAX(x) FROM source;

max
---------------------
0.93468171451240821
CREATE TEMPORARY TABLE retdemo2 (x NUMERIC);

INSERT INTO retdemo2 VALUES (random()), (random()), (random());

WITH source (average) AS (  
    SELECT AVG(x) FROM retdemo2
)
DELETE FROM retdemo2 USING source  
WHERE retdemo2.x < source.average;

SELECT * FROM retdemo2;
  x
-------------------
0.777186767663807
WITH RECURSIVE source (part_no) AS (  
    SELECT 2  
    UNION ALL  
    SELECT part.part_no  
    FROM source JOIN part ON (source.part_no = part.parent_part_no)  
)
DELETE FROM part  
USING source  
WHERE source.part_no = part.part_no;
CREATE TEMPORARY TABLE retdemo3 (x NUMERIC);

INSERT INTO retdemo3 VALUES (random()), (random()), (random());

WITH source (average) AS ( 
    SELECT AVG(x) FROM retdemo3
)
,
    source2 AS ( 
    DELETE FROM retdemo3 USING source 
    WHERE retdemo3.x < source.average
    RETURNING x
)

SELECT * FROM source2;

<table>
<thead>
<tr>
<th>x</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.185174203012139</td>
</tr>
<tr>
<td>0.209731927141547</td>
</tr>
</tbody>
</table>
Chaining Modification Commands

CREATE TEMPORARY TABLE orders (order_id SERIAL, name text);

CREATE TEMPORARY TABLE items (order_id INTEGER, part_id SERIAL, name text);

WITH source (order_id) AS ( 
    INSERT INTO orders VALUES (DEFAULT, 'my order') RETURNING order_id 
)
INSERT INTO items (order_id, name) SELECT order_id, 'my part' FROM source;

WITH source (order_id) AS ( 
    DELETE FROM orders WHERE name = 'my order' RETURNING order_id 
)
DELETE FROM items USING source WHERE source.order_id = items.order_id;
CREATE TEMPORARY TABLE old_orders (order_id INTEGER);

WITH source (order_id) AS (  
    DELETE FROM orders WHERE name = 'my order' RETURNING order_id
), source2 AS (  
    DELETE FROM items USING source WHERE source.order_id = items.order_id
)
INSERT INTO old_orders SELECT order_id FROM source;
6. Why Use CTEs

- Allows imperative processing in SQL
- Merges multiple SQL queries and their connecting application logic into a single, unified SQL query
- Improves performance by issuing fewer queries
  - reduces transmission overhead, unless server-side functions are being used
  - reduces parsing/optimizing overhead, unless prepared statements are being used
- Uses the same row visibility snapshot for the entire query, rather than requiring serializable isolation mode
- Adds an optimizer barrier between each CTE and the outer query
  - helpful with writable CTEs
  - can hurt performance when a join query is changed to use CTEs
  - optimization barrier optional in Postgres 12
Conclusion

http://momjian.us/presentations

https://www.flickr.com/photos/theophilusphotography/