The magic of WITH: Common Table Expressions

Jim Nasby, OpenSCG

https://github.com/decibel/presentations/blob/master/CTEs.pdf

- What NOT to do with CTEs
- CTEs for chaining DML
- Recursive CTEs

What not to do

- CTEs ***materialize*** everything
- WHERE clauses will not be “pushed down” to a CTE
- This means using CTEs for code clarity can seriously hurt performance

Bad

```
WITH
 tables AS (
  SELECT nspname AS table_schema, relname AS table_name
      , c.oid AS relid
    FROM pg_namespace n
      JOIN pg_class c ON n.oid = c.relnamespace
    WHERE relkind = ‘r'
  )
 , columns AS (
  SELECT table_schema, table_name, attname AS column_name
      , typname AS column_type
    FROM pg_attribute a
      JOIN pg_type type ON type.oid = atttypid
      JOIN tables tab ON tab.relid = attrelid
 )
 SELECT * FROM columns WHERE table_name = 'pg_class'
;
```

Bad

```
                     QUERY PLAN
-----------------------------------------------------
 CTE Scan on columns
   Filter: (table_name = 'pg_class'::name)
   CTE tables
     ->  Hash Join
           Hash Cond: (c.relnamespace = n.oid)
     -> Seq Scan on pg_class c
                 Filter: (relkind = 'r'::"char")
           ->  Hash
       -> Seq Scan on pg_namespace n
   CTE columns
     ->  Hash Join
           Hash Cond: (a.atttypid = type.oid)
           ->  Hash Join
                 Hash Cond: (a.attrelid = tab.relid)
       -> Seq Scan on pg_attribute a
                 ->  Hash
                       ->  CTE Scan on tables tab
           ->  Hash
       -> Seq Scan on pg_type type
(19 rows)
 Execution time: 5.693 ms
```

Good

```
SELECT table_schema, table_name
   , column_name, column_type
  FROM (
     SELECT nspname AS table_schema, relname AS table_name
         , c.oid AS relic
      FROM pg_namespace n
       JOIN pg_class c ON n.oid = c.relnamespace
      WHERE relkind = ‘r’
    ) AS tables
    JOIN (
     SELECT attname AS column_name, typname AS column_type
         , attrelid
      FROM pg_attribute a
       JOIN pg_type type ON type.oid = atttypid
    ) AS columns
     ON tables.relid = columns.attrelid
  WHERE table_name = 'pg_class'
;
```

Good

```
                                   QUERY PLAN
------------------------------------------------------------------------
--------
 Nested Loop
   ->  Nested Loop
     ->  Nested Loop
           Join Filter: (n.oid = c.relnamespace)
     ->  Index Scan using pg_class_relname_nsp_index on pg_class c
                 Index Cond: (relname = 'pg_class'::name)
                 Filter: (relkind = 'r'::"char")
     ->  Seq Scan on pg_namespace n
   ->  Index Scan using pg_attribute_... on pg_attribute a
           Index Cond: (attrelid = c.oid)
  ->  Index Scan using pg_type_oid_index on pg_type type
         Index Cond: (oid = a.atttypid)
(12 rows)
 Execution time: 0.462 ms
```

Materialization

- CTEs store their results in a “Tuple Store”
- Think temporary table, with no possibility of indexes, but a bit less overhead.

CTEs for DML

CTEs allow use of the RETURNING clause in a DML statement to do multiple things with that data

- INSERT into invoice and invoice_item tables in one statement
- DELETE from one table and INSERT data into another
- Chained UPDATE
- SELECT current data from a table and INSERT it into another table before UPDATEing the same data

Chained INSERT

```
CREATE TABLE invoice(
 invoice_id serial PRIMARY KEY
);
CREATE TABLE invoice_item(
 invoice_id int REFERENCES invoice
 , line_number int
 , item text
);
```

Chained INSERT

```
WITH invoice AS (
 INSERT INTO invoice VALUES(default)
  RETURNING invoice_id
)
INSERT INTO invoice_item(invoice_id, …)
 SELECT invoice_id, line_number, item
  FROM invoice
   -- Create two lines
   , ( VALUES
        (1, 'pizza')
        , (2, ‘soda’)
    ) line(line_number, item)
 RETURNING *
;
```

Chained INSERT

```
 invoice_id | line_number | item
------------+-------------+-------
          5 |           1 | pizza
          5 |           2 | soda
(2 rows)
```

Chained INSERT

```
WITH invoice AS(
 INSERT INTO invoice VALUES (default),(default)
  RETURNING *
)
 SELECT *
  FROM invoice
  LIMIT 1
;
```

```
 invoice_id
------------
          6
(1 row)
```

```
SELECT max(invoice_id) FROM invoice;
```

Chained INSERT

```
SELECT max(invoice_id) FROM invoice;
```

```
 max
-----
   7
(1 row)
```

CTEs always fully execute any DML operations.

Recursive CTEs

Critical to think in set theory

- Start with a single set of rows
- UNION them with a SELECT that’s self referencing
- Don’t forget the WHERE clause if necessary

Recursive CTEs

What will this output?

```
WITH RECURSIVE s AS(
 SELECT 1 AS seq
UNION ALL
 SELECT seq + 1
   FROM s
   WHERE seq <= 5
) SELECT * FROM s;
```

Recursive CTEs

```
WITH RECURSIVE s AS(
 SELECT 1 AS seq
UNION ALL
 SELECT seq + 1
   FROM s
   WHERE seq <= 5 -- Applied to input, not output
) SELECT * FROM s;
```

```
 seq
-----
   1
   2
   3
   4
   5
   6
(6 rows)
```

Recursive CTEs

```
WITH RECURSIVE s AS(
 SELECT 1 AS seq
UNION ALL
 SELECT seq + 2
   FROM s
   WHERE seq <= 5
) SELECT * FROM s;
```

Recursive CTEs

```
WITH RECURSIVE s AS(
 SELECT 1 AS seq
UNION ALL
 SELECT seq + 2
   FROM s
   WHERE seq <= 5
) SELECT * FROM s;
```

```
 seq
-----
   1
   3
   5
   7
(4 rows)
```

Recursive CTEs

```
WITH RECURSIVE s AS(
 SELECT 1 AS seq
UNION ALL
 SELECT seq + 1
   FROM s
   WHERE seq <= 5
) SELECT * FROM s;

WITH RECURSIVE s AS(
   SELECT 1 AS seq
 UNION ALL
   SELECT seq + 1 FROM s
)
 SELECT *
   FROM s
   WHERE seq <= 5
;
```

Recursive CTEs

```
                               QUERY PLAN
------------------------------------------------------------------------
 CTE Scan on s  (cost=2.95..3.57 rows=31 width=4)
   CTE s
     ->  Recursive Union  (cost=0.00..2.95 rows=31 width=4)
           ->  Result  (cost=0.00..0.01 rows=1 width=4)
           ->  WorkTable Scan on s s_1  (cost=0.00..0.23 rows=3 width=4)
                 Filter: (seq <= 5)
(6 rows)
```

```
                                QUERY PLAN
-------------------------------------------------------------------------
 CTE Scan on s  (cost=4.28..6.55 rows=34 width=4)
   Filter: (seq <= 5)
   CTE s
     ->  Recursive Union  (cost=0.00..4.28 rows=101 width=4)
           ->  Result  (cost=0.00..0.01 rows=1 width=4)
           ->  WorkTable Scan on s s_1  (cost=0.00..0.23 rows=10 width=4)
(6 rows)
```

Recursive CTEs

```
WITH RECURSIVE s AS(
 SELECT 1 AS seq
UNION ALL
 SELECT 2
UNION ALL
 SELECT seq + 1
   FROM s
   WHERE seq <= 5
) SELECT * FROM s;
```

Recursive CTEs

```
 seq
-----
   1
   2
   2
   3
   3
   4
   4
   5
   5
   6
   6
(11 rows)
```

Recursive CTEs

```
WITH RECURSIVE s AS(
 SELECT 1 AS seq, 'a' AS which
UNION ALL
 SELECT 2, 'b'
UNION ALL
 SELECT seq + 1, which
   FROM s
   WHERE seq <= 5
) SELECT * FROM s;
```

Recursive CTEs

```
 seq | which
-----+-------
   1 | a
   2 | b
   2 | a
   3 | b
   3 | a
   4 | b
   4 | a
   5 | b
   5 | a
   6 | b
   6 | a
(11 rows)
```

Recursive CTEs

```
 seq |      which
-----+------------------
   1 | a
   2 | b
   2 | a, c
   3 | b, c
   3 | a, c, c
   4 | b, c, c
   4 | a, c, c, c
   5 | b, c, c, c
   5 | a, c, c, c, c
   6 | b, c, c, c, c
   6 | a, c, c, c, c, c
(11 rows)
```

Recursive CTEs

```
WITH RECURSIVE s AS(
 SELECT 1 AS seq, 'a' AS which
UNION ALL
 SELECT 2, 'b'
UNION ALL
 SELECT seq + 1, which || ’, c’
   FROM s
   WHERE seq <= 5
) SELECT * FROM s;
```

Homework!

```
CREATE TABLE recurse(
 recurse_id serial PRIMARY KEY
 , parent_id int REFERENCES recurse
);

WITH i AS (INSERT INTO recurse(parent_id) VALUES(NULL)
RETURNING *) INSERT INTO recurse(parent_id) SELECT
recurse_id FROM i;

WITH i AS (INSERT INTO recurse(parent_id) VALUES(NULL)
RETURNING *) INSERT INTO recurse(parent_id) SELECT
recurse_id FROM i, generate_series(1,3) RETURNING *;

WITH i AS (INSERT INTO recurse(parent_id) VALUES(NULL)
RETURNING *) INSERT INTO recurse(parent_id) SELECT 5 FROM i,
generate_series(1,3) RETURNING *;
```

Homework!

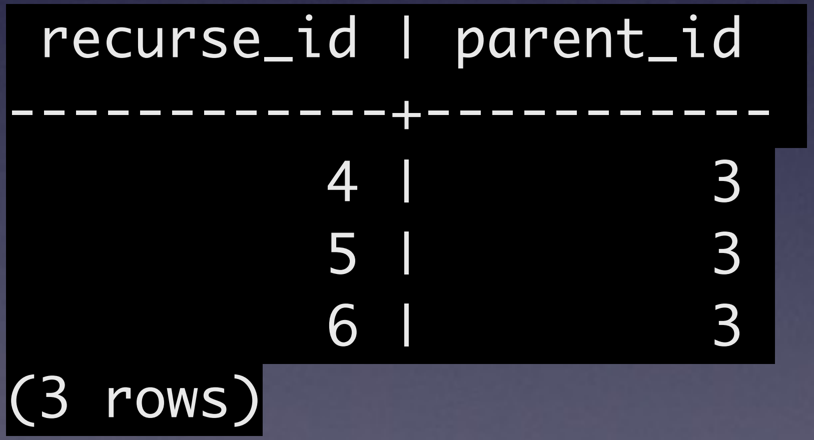
```
CREATE TABLE recurse(
 recurse_id serial PRIMARY KEY
 , parent_id int REFERENCES recurse
);

WITH i AS (INSERT INTO recurse(parent_id) VALUES(NULL)
RETURNING *) INSERT INTO recurse(parent_id) SELECT
recurse_id FROM i;
```

```
INSERT 0 1
```

How many rows are actually in the table now?

Homework!

```
WITH i AS (INSERT INTO recurse(parent_id) VALUES(NULL)
RETURNING *) INSERT INTO recurse(parent_id) SELECT
recurse_id FROM i, generate_series(1,3) RETURNING *;
```



```
 recurse_id | parent_id
------------+-----------
          4 |         3
          5 |         3
          6 |         3
(3 rows)
```

Why is 4 the starting recurse_id?

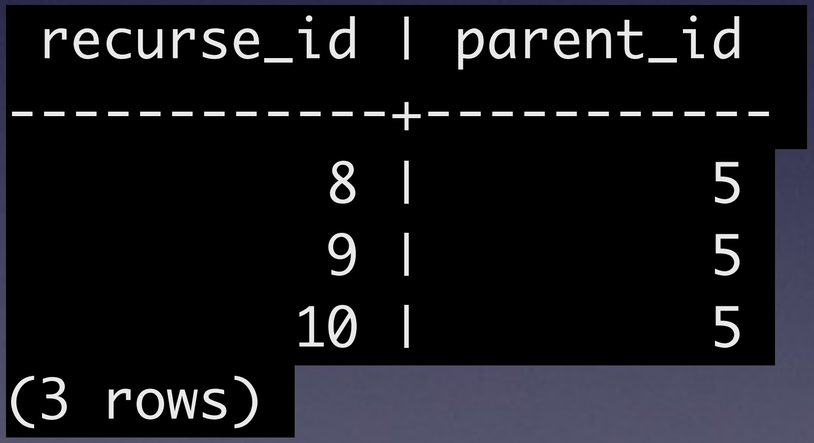
Questions?

Jim.Nasby@OpenSCG.com

https://github.com/decibel/presentations/blob/master/CTEs.pdf

Homework!

```
WITH i AS (INSERT INTO recurse(parent_id) VALUES(NULL)
RETURNING *) INSERT INTO recurse(parent_id) SELECT 5 FROM i,
generate_series(1,3) RETURNING *;
```



```
 recurse_id | parent_id
------------+-----------
          8 |         5
          9 |         5
         10 |         5
(3 rows)
```

Homework!

```
WITH RECURSIVE l AS (
  SELECT *, 1 AS level, recurse_id AS top_parent
    FROM …
```

Homework!

```
WITH RECURSIVE l AS (
  SELECT *, 1 AS level, recurse_id AS top_parent
    FROM recurse WHERE parent_id IS NULL
  UNION ALL
```

Homework!

```
WITH RECURSIVE l AS (
  SELECT *, 1 AS level, recurse_id AS top_parent
    FROM recurse WHERE parent_id IS NULL
  UNION ALL
  SELECT …
    FROM l
      JOIN recurse r ON …
```

Homework!

```
WITH RECURSIVE l AS (
  SELECT *, 1 AS level, recurse_id AS top_parent
    FROM recurse WHERE parent_id IS NULL
  UNION ALL
  SELECT …
    FROM l
      JOIN recurse r ON r.parent_id = l.recurse_id
```

Homework!

```
WITH RECURSIVE l AS (
  SELECT *, 1 AS level, recurse_id AS top_parent
    FROM recurse WHERE parent_id IS NULL
  UNION ALL
  SELECT r.recurse_id, r.parent_id, level+1, top_parent
    FROM l
      JOIN recurse r ON r.parent_id = l.recurse_id
```

Homework!

```
WITH RECURSIVE l AS (
  SELECT *, 1 AS level, recurse_id AS top_parent
    FROM recurse WHERE parent_id IS NULL
  UNION ALL
  SELECT r.recurse_id, r.parent_id, level+1, top_parent
    FROM l
      JOIN recurse r ON r.parent_id = l.recurse_id
```

```
 recurse_id | parent_id | level | top_parent 
------------+-----------+-------+------------
          1 |           |     1 |          1
          3 |           |     1 |          3
          7 |           |     1 |          7
          2 |         1 |     2 |          1
          4 |         3 |     2 |          3
          5 |         3 |     2 |          3
          6 |         3 |     2 |          3
          8 |         5 |     3 |          3
          9 |         5 |     3 |          3
         10 |         5 |     3 |          3
(10 rows)
```