Using the PostgreSQL recursive CTE to compute Bacon numbers for actors listed in the IMDb

Bryn Llewellyn
Technical Product Manager at Yugabyte

Who am I?

~

Who do I think you are?

Bryn Llewellyn

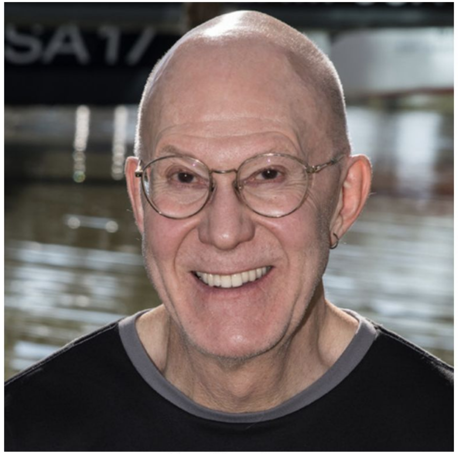
02.08.2021

Tags: postgresql database distributed sql yugabyte
Category: Interviews
Interviewed by: Andreas Scherbaum

PostgreSQL is the World's most advanced Open Source Relational Database. The interview series "PostgreSQL Person of the Week" presents the people who make the project what it is today. Read all interviews here.

LinkedIn, Twitter, and blogs

- LinkedIn: Bryn Llewellyn
- Twitter: @BrynLite
- Blogs: blog.yugabyte.com/author/bryn/



Bryn Llewellyn

Google for:

"PostgreSQL Person of the Week" Bryn

Tell us about your present job, and how it relates to PostgreSQL

I came to PostgreSQL only relatively recently, in the spring of 2019, when I left my job in the PL/SQL Team at Oracle HQ to join Yugabyte Inc. This is an exciting Silicon Valley startup, one of whose founders had been a close colleague of mine in the PL/SQL team. My first blog post in my new job, Why I Moved from Oracle to YugaByte, explains how I was easily persuaded to make this change after close to thirty years with Oracle.

- You know PostgreSQL very well
- Not a week goes by without you typing SQL at the *psql* prompt
- You don't need me to tell you about the reasons to use SQL
- You don't mind that Codd and Date laid the foundations as long ago as the nineteen-sixties
- You may, or may not, be a recursive CTE expert
- Maybe you even have some exposure to YugabyteDB

References

YugabyteDB documentation:

http://docs.yugabyte.com/latest/api/ysql/the-sql-language/with-clause/

APIs > YSQL > The SQL language >

The WITH clause and common table expressions

A `WITH` clause can be used as part of a `SELECT` statement, an `INSERT` statement, an `UPDATE` statement, or a `DELETE` statement. For this reason, the functionality is described in this dedicated section.

Companion “Bacon” blog post:

https://blog.yugabyte.com/using-postgresql-recursive-cte-part-2-bacon-numbers/

Download these slides and the companion code:

https://raw.githubusercontent.com/yugabyte/yugabyte-db/master/sample/recursive-cte-code-examples/bacon-numbers-slides-and-companion-code.zip

Recursive CTE primer / refresher

- The optional RECURSIVE keyword fundamentally changes the meaning of a CTE.

- The recursive variant lets you implement SQL solutions that, without it, at best require verbose formulations involving, for example, self-joins.

- In the limit, the recursive CTE lets you implement SQL solutions that otherwise would require procedural programming.

```
-- Syntax
with
  recursive r(c1, c2, ...) as (

    -- Non-recursive term.
    select ...

    union [all]

    -- Recursive term
    --(notice the recursive self-reference to r.
    select ... from r ...

  )

select ... from r ...;
```

- The non-recursive term is invoked just once and establishes a starting relation.

- The recursive term is invoked time and again. On its first invocation, it acts on the relation produced by the evaluation of the non-recursive term. On subsequent invocations, it acts on the relation produced by *its previous invocation*.

- Each successive term evaluation appends its output to the growing result of the recursive CTE.

- The repeating invocation of the recursive term stops when it produces an empty relation.

```sql
with
  recursive r(n) as (

    values(1)

    union all

    -- r is just the previous result
    select r.n + 1
    from r
    where r.n < 5

  )

-- r is now everything
select n from r order by n;
```

```
 n
---
 1
 2
 3
 4
 5
```

Pseudocode mental model

Purge the *final_results* table and the *previous_results* table.

Evaluate the non-recursive term, inserting the resulting rows into the *previous_results* table.

Insert the contents of the *previous_results* table into the *final_results* table.

WHILE the *previous_results* table is not empty LOOP

Purge the *temp_results* table.

Evaluate the recursive term using the current contents of the *previous_results* table for the recursive self-reference, inserting the resulting rows into the *temp_results* table.

Purge the *previous_results* table and insert the contents of the *temp_results* table.

Append the contents of the *temp_results* table into the *final_results* table.

END LOOP

Deliver the present contents of the *final_results* table.

DEMO

basics/procedural-implementation-of-recursive-cte-algorithm

From the PostgreSQL documentation:

Strictly speaking, this process
is iteration, and not recursion,
but *recursive* is the terminology (and keyword)
that the SQL Standards Committee chose.

DEMO

basics/fibonacci

The text-book use case for the recursive CTE: Traversing an employee hierarchy

Case study—using a recursive CTE to traverse an employee hierarchy

A hierarchy is a specialization of the general notion of a graph—and, as such, it's the simplest kind of graph that still deserves that name. The taxonomy of successive specializations starts with the most general (the *undirected cyclic graph*) and successively descends to the most restricted, a hierarchy. The taxonomy refers to a hierarchy as a *rooted tree*. All this is explained in the section Using a recursive CTE to traverse graphs of all kinds.

The representation of a general graph requires an explicit, distinct, representation of the nodes and the edges. Of course, a hierarchy can be represented in this way. But because of how it's restricted, it allows a simpler representation in a SQL database where only the nodes are explicitly represented, in a single table, and where the edges are inferred using a self-referential foreign key:

- A *"parent ID"* column [list] references the table's primary key—the *"ID"* column [list] enforced by a foreign key constraint.

This is referred to as a one-to-many recursive relationship, or one-to-many "pig's ear", in the jargon of entity-relationship modeling. The ultimate, unique, root of the hierarchy has the *"parent ID"* set to `NULL`.

The SQL that this section presents uses the simpler representation. But the section Using a recursive CTE to traverse graphs of all kinds shows, for completeness, that the general SQL that you need to follow edges in a general graph works when the general representation happens to describe a hierarchy. See the section Finding the paths in a rooted tree.

Download a zip of scripts that include all the code examples that implement this case study

All of the `.sql` scripts that this case study presents for copy-and-paste at the `ysqlsh` prompt are included for download in a zip-file.

Download `recursive-cte-code-examples.zip`.

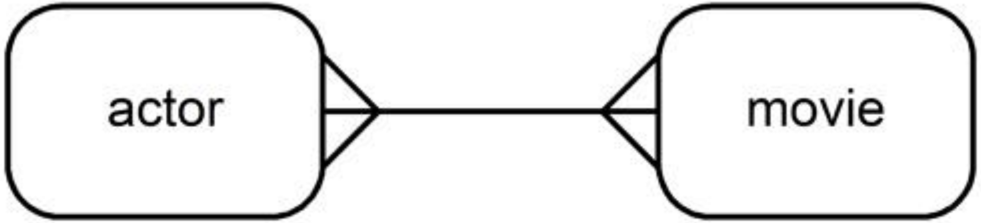
After unzipping it on a convenient new directory, you'll see a `README.txt`. It tells you how to start the master-script. Simply start it in `ysqlsh`. You can run it time and again. It always finishes silently. You can see the report that it produces on a dedicated spool directory and confirm that your report is identical to the reference copy that is delivered in the zip-file.

DEMO

employee-hierarchy/
  0.sql
  2-bare-recursive-cte.sql

```
recursive hierarchy_of_emps(depth, mgr_name, name) as (
  -- Non-recursive term.
  -- Select the exactly one ultimate manager.
  -- Define this emp to be at depth 1.
  (
    select
      1,
      '-',
      name
    from emps
    where mgr_name is null
  )
  union all
  -- Recursive term.
  -- Treat the emps from the previous iteration as managers.
  -- Join these with their reports, if they have any.
  -- Increase the emergent depth by 1 with each step.
  -- Stop when none of the current putative managers has a report.
  -- Each successive iteration goes one level deeper in the hierarchy.
  (
    select
      h.depth + 1,
      e.mgr_name,
      e.name
    from
    emps as e
    inner join
    hierarchy_of_emps as h on e.mgr_name = h.name
  )
)
```

Bacon numbers using synthetic data



Implies these tables:

```
actors(actor text primary key)

movies(movie text primary key)

cast_members(
  actor text, movie text,
  constraint cast_members_pk primary key(actor, movie)
  ...
```

```
insert into actors(actor) values
  ('Alfie'),
  ('Chloe'),
  ('Emily'),
  ('Helen'),
  ('James'),
  ('Steve');
```

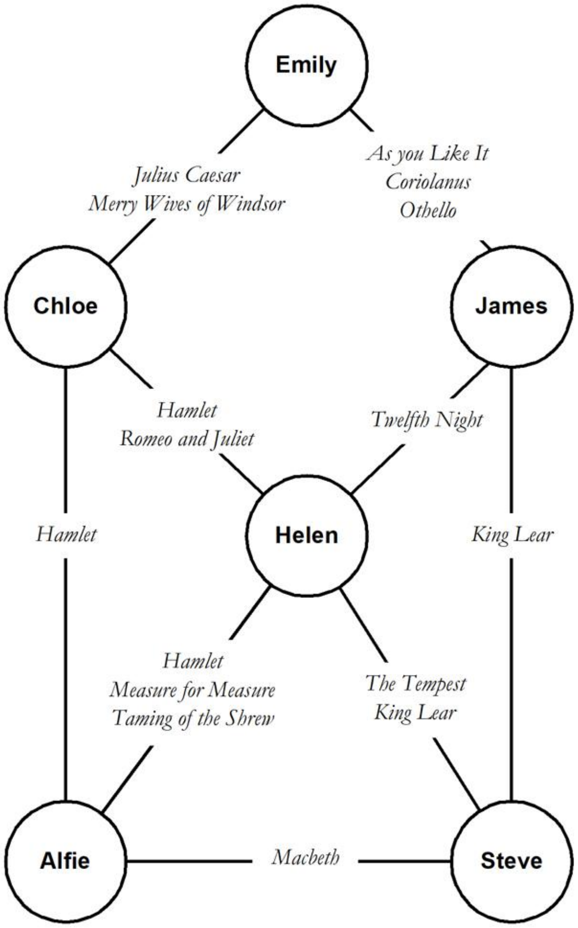
```
insert into movies(movie) values
  ('As You Like It'),
  ('Coriolanus'),
  ('Hamlet'),
  ('Julius Caesar'),
  ('King Lear'),
  ('Macbeth'),
  ('Measure for Measure'),
  ('Merry Wives of Windsor'),
  ('Othello'),
  ('Romeo and Juliet'),
  ('Taming of the Shrew'),
  ('The Tempest'),
  ('Twelfth Night');
```

```
insert into cast_members(actor, movie) values

  ( 'Alfie'   ,  'Hamlet'                  ),
  ( 'Alfie'   ,  'Macbeth'                 ),
  ( 'Alfie'   ,  'Measure for Measure'     ),
  ( 'Alfie'   ,  'Taming of the Shrew'     ),

  ( 'Helen'   ,  'The Tempest'             ),
  ( 'Helen'   ,  'Hamlet'                  ),
  ( 'Helen'   ,  'King Lear'               ),
  ( 'Helen'   ,  'Measure for Measure'     ),
  ( 'Helen'   ,  'Romeo and Juliet'        ),
  ( 'Helen'   ,  'Taming of the Shrew'     ),
  ( 'Helen'   ,  'Twelfth Night'           ),

  ( 'Emily'   ,  'As You Like It'          ),
  ( 'Emily'   ,  'Coriolanus'              ),
  ( 'Emily'   ,  'Julius Caesar'           ),
  ( 'Emily'   ,  'Merry Wives of Windsor'  ),
  ( 'Emily'   ,  'Othello'                 ),

  ( 'Chloe'   ,  'Hamlet'                  ),
  ( 'Chloe'   ,  'Julius Caesar'           ),
  ( 'Chloe'   ,  'Merry Wives of Windsor'  ),
  ( 'Chloe'   ,  'Romeo and Juliet'        ),

  ( 'James'   ,  'As You Like It'          ),
  ( 'James'   ,  'Coriolanus'              ),
  ( 'James'   ,  'King Lear'               ),
  ( 'James'   ,  'Othello'                 ),
  ( 'James'   ,  'Twelfth Night'           ),

  ( 'Steve'   ,  'The Tempest'             ),
  ( 'Steve'   ,  'King Lear'               ),
  ( 'Steve'   ,  'Macbeth'                 );
```

Implies this mechanically populated table:

```
edges(
  node_1 text,
  node_2 text,
  movies text[],

  constraint edges_pk primary key(node_1, node_2),
  constraint edges_fk_1 foreign key(node_1)
    references actors(actor),
  constraint edges_fk_2 foreign key(node_2)
    references actors(actor))
```

```
node_1 | node_2 |                        movies
-------+--------+--------------------------------------------------------
Alfie  | Chloe  | Hamlet
Alfie  | Helen  | Hamlet | Measure for Measure | Taming of the Shrew
Alfie  | Steve  | Macbeth
Chloe  | Emily  | Julius Caesar | Merry Wives of Windsor
Chloe  | Helen  | Hamlet | Romeo and Juliet
Emily  | James  | As You Like It | Coriolanus | Othello
Helen  | James  | King Lear | Twelfth Night
Helen  | Steve  | King Lear | The Tempest
James  | Steve  | King Lear
```


Emily
Julius Caesar
Merry Wives of Windsor
As you Like It
Coriolanus
Othello
Chloe
James
Hamlet
Romeo and Juliet
Twelfth Night
Hamlet
Helen
King Lear
Hamlet
Measure for Measure
Taming of the Shrew
The Tempest
King Lear
Alfie
Macbeth
Steve

```
procedure find_paths(start_node in text)
  language plpgsql
as $body$
begin
  ...
  recursive paths(path) as (
    select array[start_node, node_2]
    from edges
    where node_1 = start_node

    union all

    select p.path||e.node_2
    from edges e
    inner join paths p on e.node_1 = terminal(p.path)
    where not e.node_2 = any(p.path) -- <<<<< Prevent cycles.
    )
  ...
```

DEMO

bacon-numbers/
  03-insert-synthetic-data.sql
  04-populate-edges-table-and-check-contents.sql
  07-find-paths-naive.sql

1	2	Emily > Chloe
2	2	Emily > James
3	3	Emily > Chloe > Alfie
4	3	Emily > Chloe > Helen
5	3	Emily > James > Helen
6	3	Emily > James > Steve
7	4	Emily > Chloe > Alfie > Helen
8	4	Emily > Chloe > Alfie > Steve
9	4	Emily > Chloe > Helen > Alfie
10	4	Emily > Chloe > Helen > James
11	4	Emily > Chloe > Helen > Steve
12	4	Emily > James > Helen > Alfie
13	4	Emily > James > Helen > Chloe
14	4	Emily > James > Helen > Steve
15	4	Emily > James > Steve > Alfie
16	4	Emily > James > Steve > Helen
17	5	Emily > Chloe > Alfie > Helen > James
18	5	Emily > Chloe > Alfie > Helen > Steve
19	5	Emily > Chloe > Alfie > Steve > Helen
20	5	Emily > Chloe > Alfie > Steve > James
21	5	Emily > Chloe > Helen > Alfie > Steve
22	5	Emily > Chloe > Helen > James > Steve
23	5	Emily > Chloe > Helen > Steve > Alfie
24	5	Emily > Chloe > Helen > Steve > James
25	5	Emily > James > Helen > Alfie > Chloe
26	5	Emily > James > Helen > Alfie > Steve
27	5	Emily > James > Helen > Chloe > Alfie
28	5	Emily > James > Helen > Steve > Alfie
29	5	Emily > James > Steve > Alfie > Chloe
30	5	Emily > James > Steve > Alfie > Helen
31	5	Emily > James > Steve > Helen > Alfie
32	5	Emily > James > Steve > Helen > Chloe
33	6	Emily > Chloe > Alfie > Helen > James > Steve
34	6	Emily > Chloe > Alfie > Helen > Steve > James
35	6	Emily > Chloe > Alfie > Steve > Helen > James
36	6	Emily > Chloe > Alfie > Steve > James > Helen
37	6	Emily > Chloe > Helen > Alfie > Steve > James
38	6	Emily > Chloe > Helen > James > Steve > Alfie
39	6	Emily > James > Helen > Chloe > Alfie > Steve
40	6	Emily > James > Helen > Steve > Alfie > Chloe
41	6	Emily > James > Steve > Alfie > Chloe > Helen
42	6	Emily > James > Steve > Alfie > Helen > Chloe
43	6	Emily > James > Steve > Helen > Alfie > Chloe
44	6	Emily > James > Steve > Helen > Chloe > Alfie

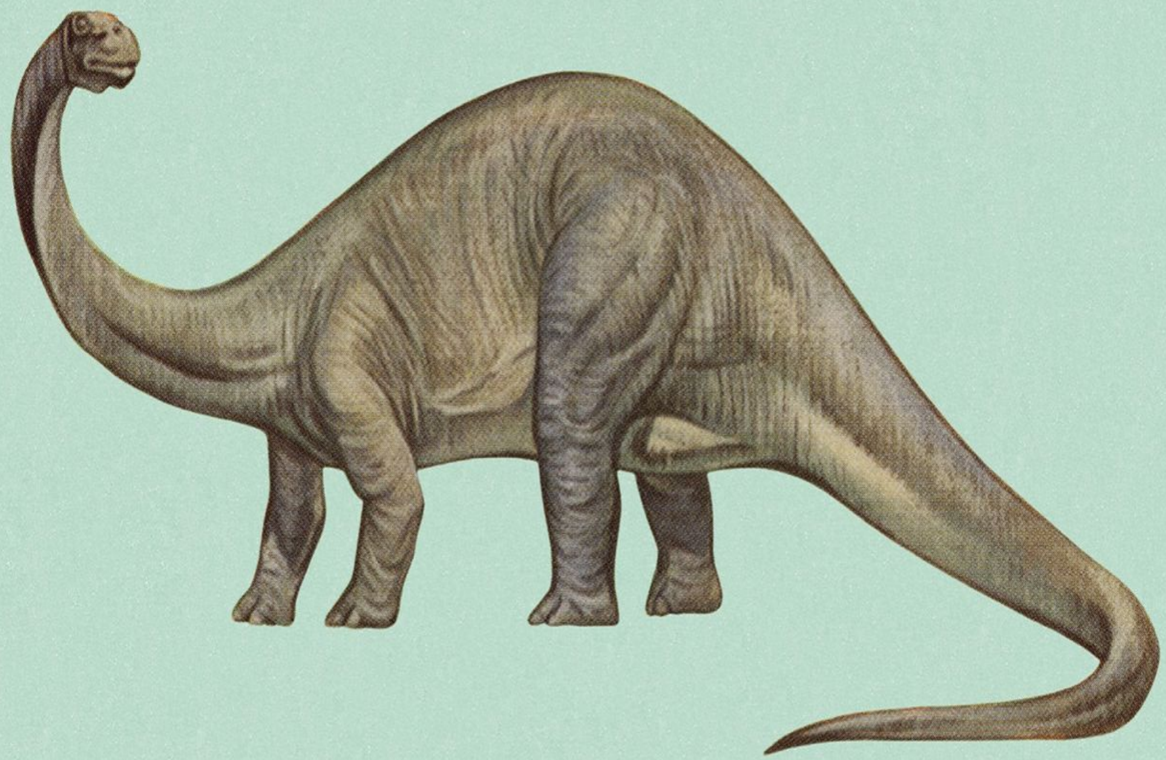
Jonathan Lewis[1] coined the term "Brontosaurus query" for one that produces, during its execution, very many more results than are eventually retained. The visual metaphor speaks for itself.

See, for example, his talk "Just Don't Do It Sins of omission and commission"[2]. He shows that a hybrid solution that uses a stored procedure to execute several SQL statements is sometimes the only way to tame the beast.

[1] https://www.linkedin.com/in/jonathan-lewis-34093a2a/

[2] https://acesathome.com/wp-content/uploads/2020/06/just_dont_do_it_aces.pdf

Beware the Brontosaurus query!



```sql
-- Emulate the non-recursive term.
delete from raw_paths;
delete from previous_paths;

insert into previous_paths(path)
select array[start_node, e.node_2]
from edges e
where e.node_1 = start_node;

insert into raw_paths(path)
select r.path from previous_paths r;
```

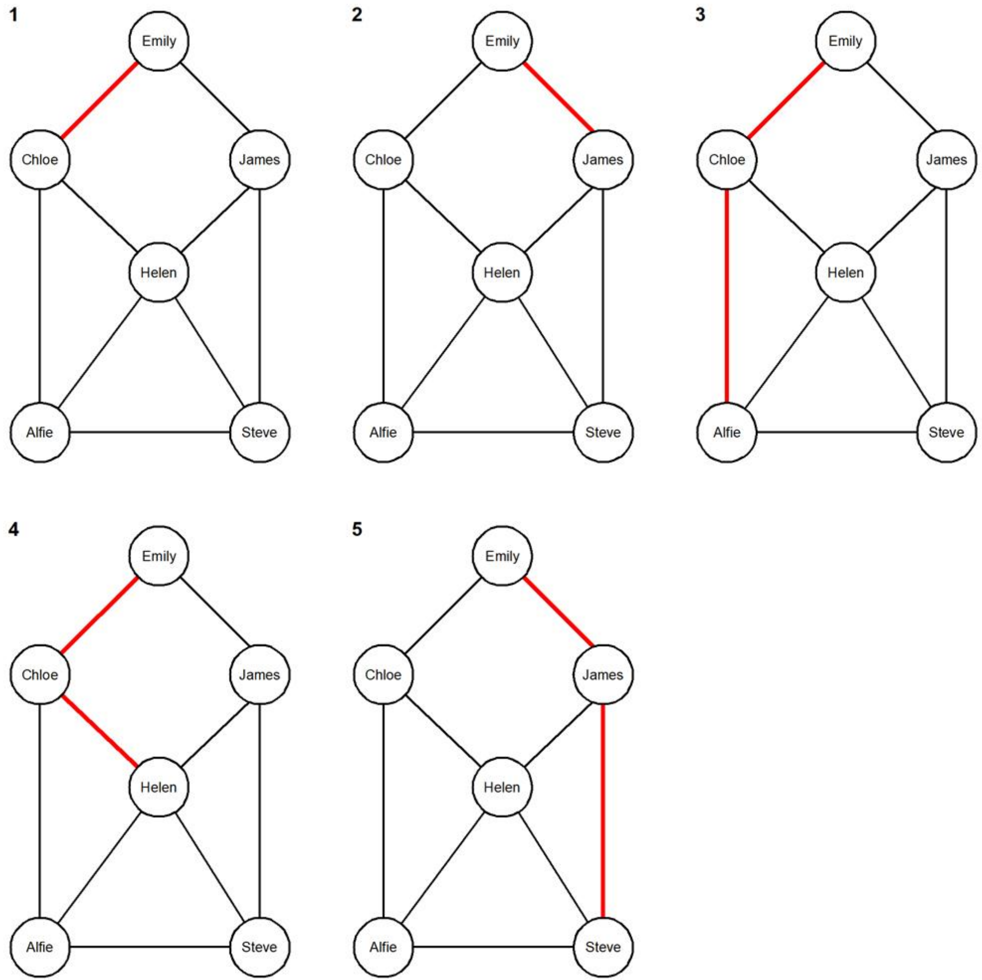
```
-- Emulate the recursive term.
loop
  delete from temp_paths;
  insert into temp_paths(path)
  select w.path||e.node_2
  from edges e
  inner join previous_paths w on e.node_1 = terminal(w.path)
  where not e.node_2 = any(w.path); -- <<<<< Prevent cycles.

  get diagnostics n = row_count;
  exit when n < 1;

  if prune then
    ...
  end if;

  delete from previous_paths;
  insert into previous_paths(path) select t.path from temp_paths t;
  insert into raw_paths (path) select t.path from temp_paths t;
end loop;
```

```
if prune then
  delete from temp_paths
  where
  (
    -- Prune all but one path to each distinct new terminal.
    path not in (select min(path) from temp_paths group by terminal(path))
  )
  or
  (
    -- Prune newer (and therefore longer) paths to
    -- already-found terminals.
    terminal(path) in
    (
      select terminal(path)
      from raw_paths
    )
  );
end if;
```

DEMO

bacon-numbers/
  09-find-paths-no-pruning.sql
  11-find-paths-with-pruning.sql

1	2	Emily > Chloe
2	2	Emily > James
3	3	Emily > Chloe > Alfie
4	3	Emily > Chloe > Helen
5	3	Emily > James > Steve



DEMO

bacon-numbers/
12-cr-decorated-paths-report.sql

```
Emily
  Julius Caesar
  Merry Wives of Windsor
      Chloe
--------------------------------------------------
Emily
  As You Like It
  Coriolanus
  Othello
      James
--------------------------------------------------
Emily
  Julius Caesar
  Merry Wives of Windsor
      Chloe
        Hamlet
            Alfie
--------------------------------------------------
Emily
  Julius Caesar
  Merry Wives of Windsor
      Chloe
        Hamlet
        Romeo and Juliet
            Helen
--------------------------------------------------
Emily
  As You Like It
  Coriolanus
  Othello
      James
        King Lear
            Steve
```

DEMO

bacon-numbers/
 99-pruning-demo.sql

"raw_paths" to date after one rep. of the recursive term.

```
{Emily,Chloe,Alfie}
{Emily,Chloe,Helen}
{Emily,Chloe}
{Emily,James,Steve}
{Emily,James}
```

"temp_paths" produced by the second rep. of the recursive term before pruning.

```
{Emily,Chloe,Alfie,Helen}
{Emily,Chloe,Alfie,Steve}
{Emily,Chloe,Helen,Alfie}
{Emily,Chloe,Helen,James}
{Emily,Chloe,Helen,Steve}
{Emily,James,Helen,Alfie}
{Emily,James,Helen,Chloe}
{Emily,James,Helen,Steve}
{Emily,James,Steve,Alfie}
{Emily,James,Steve,Helen}
```

"temp_paths" after pruning all but one path to each distinct new terminal.

```
{Emily,Chloe,Alfie,Helen}
{Emily,Chloe,Alfie,Steve}
{Emily,Chloe,Helen,Alfie}
{Emily,Chloe,Helen,James}
{Emily,James,Helen,Chloe}
```

"temp_paths" after pruning newer (and therefore longer) paths to already-found terminals.

Nothing survives. So the (so-called) recursion stops.

Bacon numbers using a curated subset of the real IMDb data

```
count(*) from cast_members...  1,817

count(*) from actors.........    161

count(*) from movies.........  1,586
```

DEMO

bacon-numbers/
13-insert-imdb-data.sql
14-inspect-imdb-data.sql
15-find-imdb-paths.sql

```
Seed: Kevin Bacon
total number of pruned paths:           160
Max path length:                        6
unreached:                              Kevin Bacon

Kevin Bacon
  She's Having a Baby (1988)
  Wild Things (1998)
    Bill Murray
      Saturday Night Live: Game Show Parodies (1998)
        Billy Crystal
          Muhammad Ali: Through the Eyes of the World (2001)
            James Earl Jones
              Looking for Richard (1996)
                Al Pacino
                  Christopher Nolan Interviews Al Pacino (2002)
                    Christopher Nolan
```

```
Seed: Christopher Nolan
total number of pruned paths:          160
Max path length:                       6
unreached:                             Christopher Nolan

Christopher Nolan
  Christopher Nolan Interviews Al Pacino (2002)
      Al Pacino
        Looking for Richard (1996)
            James Earl Jones
              Conan Unchained: The Making of 'Conan' (2000)
              Conan the Barbarian (1982)
                  Arnold Schwarzenegger
                    Last Party, The (1993)
                        Christian Slater
                          Murder in the First (1995)
                              Kevin Bacon
```

Summary

- **The recursive CTE is a very powerful SQL feature that supports, *inter alia*, finding paths in graphs of all kinds: general undirected cyclic graph; directed cyclic graph; directed acyclic graph; and rooted tree. (The much-loved employee hierarchy is a rooted tree.)**

- **But it finds *all* paths. There's no way to express "find only one among each of the shortest paths to each distinct reachable nodes".**

- **A hybrid PL/pgSQL-SQL approach comes to the rescue because it allows intervention to implement early pruning of each newly-discovered uninteresting path.**

- **The hybrid approach allows an acceptable quick solution to the famous Bacon numbers problem on real IMDb data.**

Finally...

Most Advanced Open Source Distributed SQL

PostgreSQL

Query Layer

World's Most Advanced
Open Source SQL Engine

Google Spanner

Storage Layer

World's Most Advanced
Distributed OLTP Architecture

Reuse

Inspiration

yugabyteDB

Questions?

Download
download.yugabyte.com

Join Slack Discussions
yugabyte.com/slack

Star on GitHub
github.com/yugabyte/yugabyte-db