Recursive Queries

in PostgreSQL 8.4

Gregory Stark, EnterpriseDB UK Ltd.

EnterpriseDB™

Thank You!

Earlier Work

```
From: Evgen Potemkin <evgent@ns.terminal.ru>
Subject: Proposal of hierachical queries (a la Oracle)
Date: 2002-11-14 11:52:28 GMT (6 years, 27 weeks, 14 hours and 29 minutes ago)

 Hi there!

 I want to propose the patch for adding the hierarchical queries
 posibility. It allows to construct queries a la Oracle for ex:
 SELECT a,b FROM t CONNECT BY a PRIOR b START WITH cond;B

 I've seen this type of queries often made by adding a new type, which
 stores position of row in the tree. But sorting such tree are very
 tricky (i think).

 Patch allows result tree to be sorted, i.e. subnodes of each node will
 be sorted by ORDER BY clause.

 with regards, evgen
```

Thank You!

Finally in 2008 Yoshiyuki Asaba-san steps forward:

```
From: Tatsuo Ishii <ishii@postgresql.org>
Subject: RFP: Recursive query in 8.4
Date: 2008-02-19 08:36:00 GMT (1 year, 12 weeks, 6 days ago)

Hi,

As I promised before we would like to propose implementing the
recursive query as defined in the standard for PostgreSQL 8.4.

The work is supported by Sumitomo Electric Information Systems Co.,
Ltd. (http://www.sei-info.co.jp/) and SRA OSS, Inc. Japan
(http://www.sraoss.co.jp).
```

Reviewers etc:

David Fetter
Markus Wanner
Zoltan Boszormenyi
Hans-Juergen Schoenig
Jeff Davis

Thank You!

And finally:

```
commit bc5d2de37c5dd99c2057763fe24174dee1ee161b
Author: Tom Lane <tgl@sss.pgh.pa.us>
Date:   Sat Oct 4 21:56:55 2008 +0000

Implement SQL-standard WITH clauses, including WITH RECURSIVE.

There are some unimplemented aspects: recursive queries must use UNION
ALL (should allow UNION too), and we don't have SEARCH or CYCLE clauses.
These might or might not get done for 8.4, but even without them it's a
pretty useful feature.

There are also a couple of small loose ends and definitional quibbles,
which I'll send a memo about to pgsql-hackers shortly. But let's land
the patch now so we can get on with other development.

Yoshiyuki Asaba, with lots of help from Tatsuo Ishii and Tom Lane
```

Non-recursive WITH

```
  WITH active_users
    AS (SELECT * FROM users WHERE active)
SELECT *
  FROM active_users
```

- cheap temporary table within a query
- convenient short-cut for writing queries
- useful for avoiding repeated work

Non-recursive WITH

```
  WITH users_sample
    AS (SELECT *
          FROM users
         WHERE random()< 0.1)
SELECT a.*
  FROM users_sample as a
  JOIN users_sample as b
    ON (a.parent = b.id)
```

WITH guarantees that the query will only be evaluated once. This avoids duplicated work and guarantees the same sample, but it also means indexes on users can't be used to perform the join.

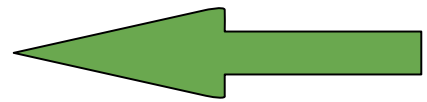
Recursion Basics

Recursive algorithms and data structures always have two parts:

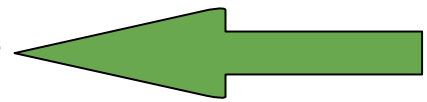
- Base Case
- Recursion

Recursive WITH

```
WITH RECURSIVE cte AS (
      SELECT 1 AS i
    UNION ALL
      SELECT i+1 from cte
       WHERE i < 10
    )
SELECT *
  FROM cte
```



Base Case



Recursion

The "RECURSIVE" keyword is mandatory. Otherwise "cte" in the WITH clause would refer to a table, view, or alias in an outer level of the query.

Danger: Infinite Recursion

```
WITH RECURSIVE cte AS (
      SELECT 1 AS i
   UNION ALL
      SELECT i+1 from cte
       WHERE i < 10
    )
SELECT *
   FROM cte
 LIMIT 10
```

What is a query anyways?

- SQL queries are not procedural
- Queries describe the requested data
- Queries mirror the data structures in the database
- So the question is: What kinds of data structures do recursive queries describe?

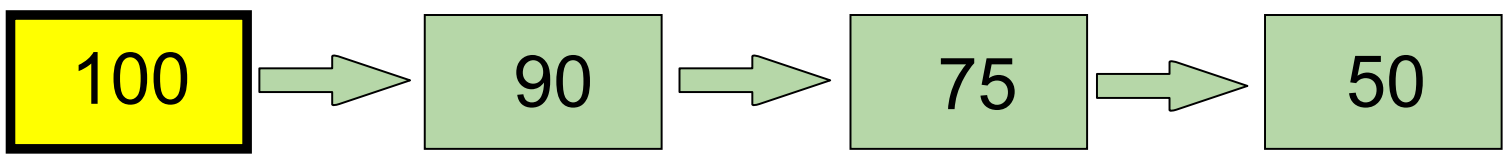
What kinds of data structures?

Recursive data structures of course!

- Linked Lists
- Binary Trees
- Graphs

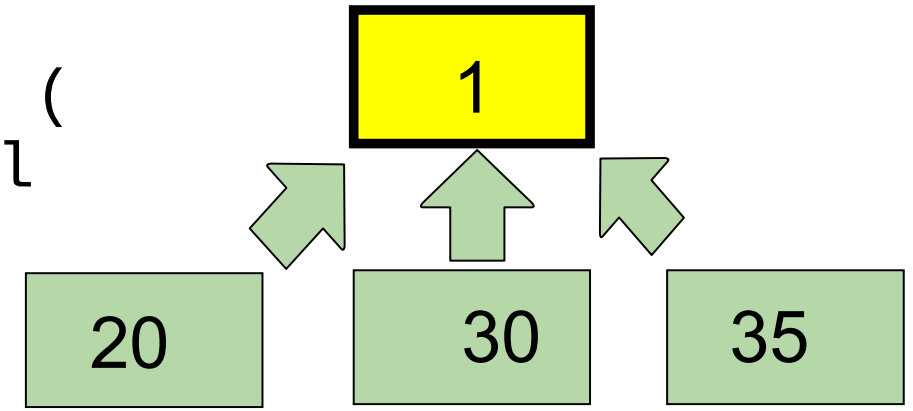
Linked Lists

```
WITH RECURSIVE thread AS (
      SELECT *
        FROM msgs
       WHERE msg_id = 100
  UNION ALL
      SELECT msgs.*
        FROM thread
        JOIN msgs ON (msgs.msg_id = thread.in_reply_to)
  )
SELECT *
  FROM thread;
```



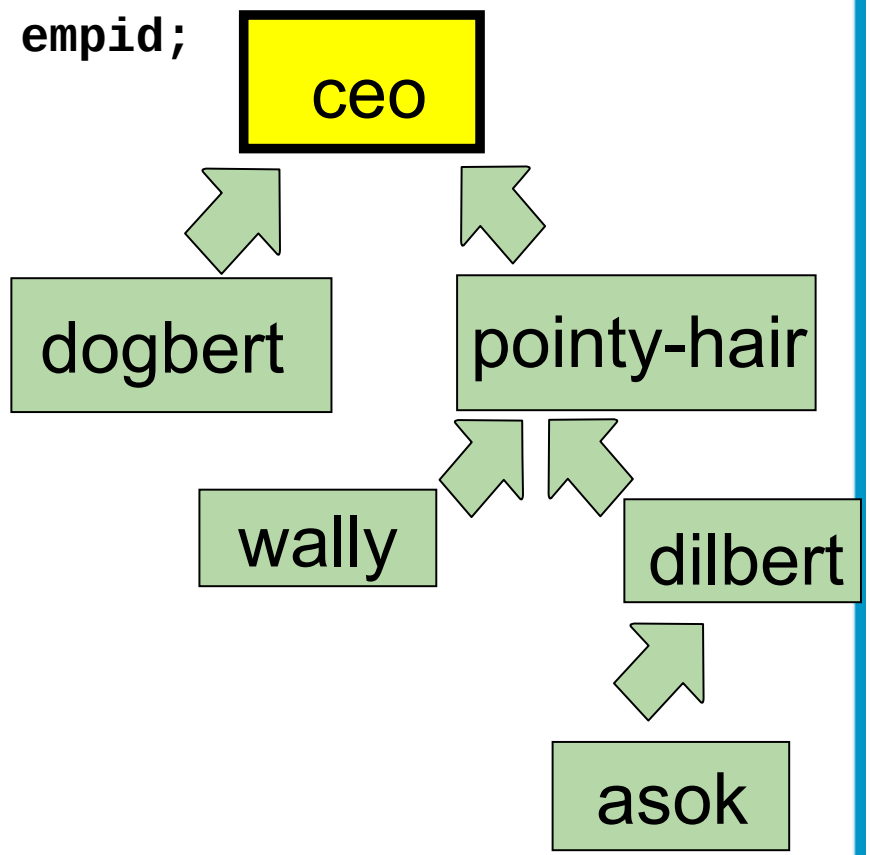
Trees

```
WITH RECURSIVE thread AS (
      SELECT *, 0 as level
        FROM msgs
       WHERE msg_id = 1
   UNION ALL
      SELECT msgs.*, level+1
        FROM thread
        JOIN msgs ON (thread.msg_id = msgs.in_reply_to)
  )
SELECT *
  FROM thread;
```



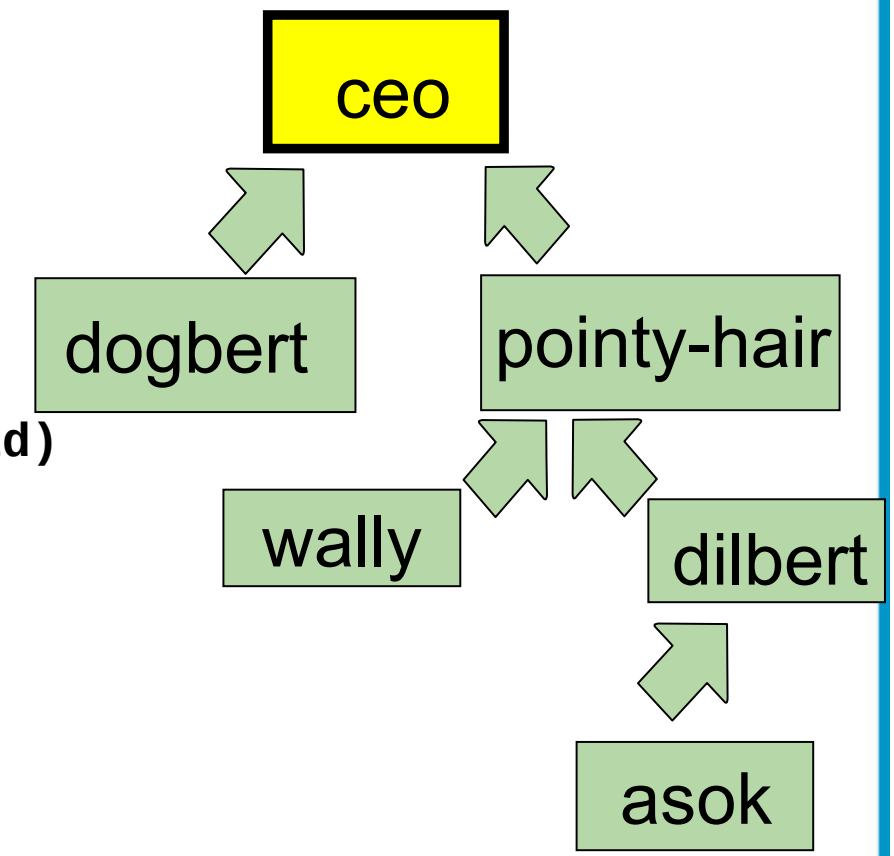
Trees

```
postgres=# select * from emp order by empid;
 empid | mgrid |    name
-------+-------+-------------
     0 |       | ceo
     1 |     0 | pointy-hair
     2 |     0 | dogbert
     3 |     1 | dilbert
     4 |     1 | wally
     5 |     3 | asok
(6 rows)
```



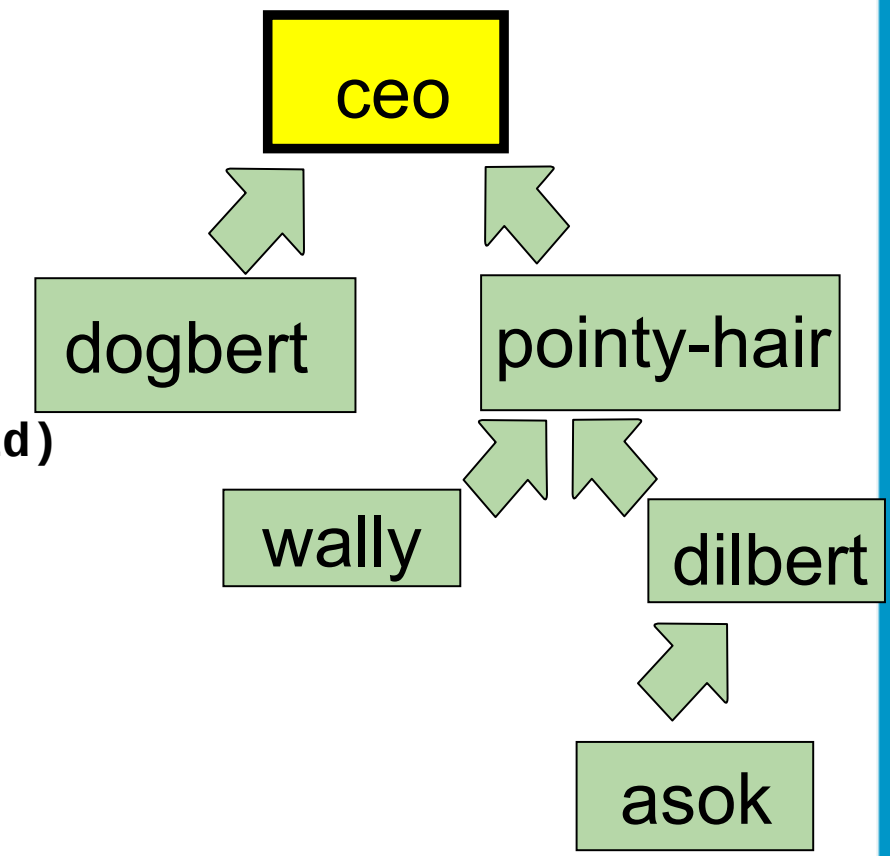
Trees

```
WITH RECURSIVE x AS (
   SELECT *
     FROM emp
    WHERE empid=5
UNION ALL
   SELECT emp.*
     FROM x
     JOIN emp ON (emp.empid = x.mgrid)
  )
SELECT * FROM x;

 empid | mgrid |    name
-------+-------+-------------
     5 |     3 | asok
     3 |     1 | dilbert
     1 |     0 | pointy-hair
     0 |       | ceo
(4 rows)
```



Trees

```
WITH RECURSIVE x AS (
   SELECT *
     FROM emp
    WHERE empid=1
UNION ALL
   SELECT emp.*
     FROM x
     JOIN emp ON (emp.mgrid = x.empid)
 )
SELECT * FROM x;

 empid | mgrid |    name
-------+-------+-------------
     1 |     0 | pointy-hair
     3 |     1 | dilbert
     4 |     1 | wally
     5 |     3 | asok
(4 rows)
```



Unsupported Features

```
WITH cte AS ()

  SEARCH {DEPTH|BREADTH} FIRST BY <sort specification list>
     SET sequence_column

   CYCLE col1,... SET <cycle mark column> TO <expression>
 DEFAULT <expression> USING <column>
```

Postgres doesn't support DEPTH FIRST or BREADTH FIRST. The implementation effectively always does breadth first.

Postgres doesn't support CYCLE to detect loops

Workarounds

```
WITH RECURSIVE  x AS (
   SELECT *, 0 AS n, array[empid] AS chain
     FROM emp
    WHERE empid=0
UNION ALL
   SELECT emp.*, n+1, chain||emp.empid
     FROM x
     JOIN emp on (emp.mgrid = x.empid)
    WHERE NOT emp.empid = ANY (chain)
 )
select * from x
order by chain
```

detect cycles

depth-first order

```
     0 |       | ceo          | 0 | {0}
     1 |     0 | pointy-hair  | 1 | {0,1}
     3 |     1 | dilbert      | 2 | {0,1,3}
     5 |     3 | asok         | 3 | {0,1,3,5}
     4 |     1 | wally        | 2 | {0,1,4}
     2 |     0 | dogbert      | 1 | {0,2}
```

Workarounds

```
WITH RECURSIVE  x AS (
   SELECT *, 0 AS n, array[ROW(name,empid)] AS chain
     FROM emp
    WHERE empid=0
UNION ALL
   SELECT emp.*, n+1, chain||ROW(emp.name,emp.empid)
     FROM x
     JOIN emp ON (emp.mgrid = x.empid)
 )
SELECT *
  FROM x
 ORDER BY chain
```

Simulate Oracle-style ORDER SIBLINGS BY using an array of ROW()s containing the sort key.

Note that Postgres still does a breadth-first search and then sorts afterwards.

```
 0 |      | ceo         | 0 | {"(ceo,0)"}
 2 |    0 | dogbert     | 1 | {"(ceo,0)","(dogbert,2)"}
 1 |    0 | pointy-hair | 1 | {"(ceo,0)","(pointy-hair,1)"}
 3 |    1 | dilbert     | 2 | {"(ceo,0)","(pointy-hair,1)","(dilbert,3)"}
 5 |    3 | asok        | 3 | {"(ceo,0)","(pointy-hair,1)","(dilbert,3)","(asok,5)"}
 4 |    1 | wally       | 2 | {"(ceo,0)","(pointy-hair,1)","(wally,4)"}
```

Explaining Recursive Queries

```
EXPLAIN WITH RECURSIVE x AS (
   SELECT *
     FROM emp
    WHERE empid=0
UNION ALL
   SELECT emp.*
     FROM x
     JOIN emp ON (emp.mgrid = x.empid)
 )
SELECT * FROM x;

                                    QUERY PLAN
-------------------------------------------------------------------------------
 CTE Scan on x  (cost=408.02..477.74 rows=3486 width=40)
   CTE x
     ->  Recursive Union  (cost=0.00..408.02 rows=3486 width=40)
           ->  Seq Scan on emp  (cost=0.00..24.50 rows=6 width=40)
                 Filter: (empid = 0)
           ->  Hash Join  (cost=1.95..31.38 rows=348 width=40)
                 Hash Cond: (public.emp.mgrid = x.empid)
                 ->  Seq Scan on emp  (cost=0.00..21.60 rows=1160 width=40)
                 ->  Hash  (cost=1.20..1.20 rows=60 width=4)
                       ->  WorkTable Scan on x  (cost=0.00..1.20 rows=60 width=4)
```

Useful examples

PostgreSQL internal dependency graph. Explains what objects a DROP command will cascade to:

```
postgres=# select * from pg_depend
            where refclassid = 'pg_class'::regclass
              and refobjid = 'messages'::regclass;

 classid | objid | objsubid | refclassid | refobjid | refobjsubid | deptype
---------+-------+----------+------------+----------+-------------+---------
    1259 | 16399 |        0 |       1259 |    16395 |           0 | i
    1247 | 16397 |        0 |       1259 |    16395 |           0 | i
    2606 | 16404 |        0 |       1259 |    16395 |           1 | n
    1259 | 16403 |        0 |       1259 |    16395 |           1 | a
    1259 | 16393 |        0 |       1259 |    16395 |           1 | a
    2604 | 16398 |        0 |       1259 |    16395 |           1 | a
    2606 | 16404 |        0 |       1259 |    16395 |           2 | a
(7 rows)
```

Useful examples

```
WITH RECURSIVE tree AS (
    SELECT 'messages'::regclass::text AS tree,
           0 AS level,
           'pg_class'::regclass AS classid,
           'messages'::regclass AS objid
 UNION ALL
    SELECT tree ||
             ' <-- ' ||
             get_obj_description(pg_depend.classid, pg_depend.objid),
           level+1,
           pg_depend.classid,
           pg_depend.objid
      FROM tree
      JOIN pg_depend ON (  tree.classid = pg_depend.refclassid
                       AND tree.objid = pg_depend.refobjid)
)
SELECT tree.tree
  FROM tree
 WHERE level < 10
```

Useful examples

```
                                                tree
-------------------------------------------------------------------------------------------
 messages
 messages <-- CONSTRAINT in_reply_to_fkey
 messages <-- CONSTRAINT in_reply_to_fkey
 messages <-- CONSTRAINT in_reply_to_fkey <-- TRIGGER RI_ConstraintTrigger_16405
 messages <-- CONSTRAINT in_reply_to_fkey <-- TRIGGER RI_ConstraintTrigger_16405
 messages <-- CONSTRAINT in_reply_to_fkey <-- TRIGGER RI_ConstraintTrigger_16406
 messages <-- CONSTRAINT in_reply_to_fkey <-- TRIGGER RI_ConstraintTrigger_16406
 messages <-- CONSTRAINT in_reply_to_fkey <-- TRIGGER RI_ConstraintTrigger_16407
 messages <-- CONSTRAINT in_reply_to_fkey <-- TRIGGER RI_ConstraintTrigger_16407
 messages <-- CONSTRAINT in_reply_to_fkey <-- TRIGGER RI_ConstraintTrigger_16408
 messages <-- CONSTRAINT in_reply_to_fkey <-- TRIGGER RI_ConstraintTrigger_16408
 messages <-- DEFAULT messages.message_id
 messages <-- TYPE messages
 messages <-- TYPE messages <-- TYPE messages[]
 messages <-- idx_msgs
 messages <-- idx_msgs <-- CONSTRAINT in_reply_to_fkey
 messages <-- idx_msgs <-- CONSTRAINT in_reply_to_fkey <-- TRIGGER RI_ConstraintTrigger_16405
 messages <-- idx_msgs <-- CONSTRAINT in_reply_to_fkey <-- TRIGGER RI_ConstraintTrigger_16406
 messages <-- idx_msgs <-- CONSTRAINT in_reply_to_fkey <-- TRIGGER RI_ConstraintTrigger_16407
 messages <-- idx_msgs <-- CONSTRAINT in_reply_to_fkey <-- TRIGGER RI_ConstraintTrigger_16408
 messages <-- messages_message_id_seq
 messages <-- messages_message_id_seq <-- DEFAULT messages.message_id
 messages <-- messages_message_id_seq <-- TYPE messages_message_id_seq
 messages <-- pg_toast.pg_toast_16395
 messages <-- pg_toast.pg_toast_16395 <-- TYPE pg_toast.pg_toast_16395
 messages <-- pg_toast.pg_toast_16395 <-- pg_toast.pg_toast_16395_index
 messages <-- pg_toast.pg_toast_16395 <-- pg_toast.pg_toast_16395_index
(27 rows)
```

Not so useful examples...

```
WITH RECURSIVE z(ix, iy, cx, cy, x, y, i) AS (
   SELECT ix, iy, x::float, y::float, x::float, y::float, 0
     FROM (select -1.88+0.016*i, i from generate_series(0,150) as i) as xgen(x,ix),
          (select -1.11+0.060*i, i from generate_series(0,36)  as i) as ygen(y,iy)
    UNION ALL
   SELECT ix, iy, cx, cy, x*x - y*y + cx, y*x*2 + cy, i+1
     FROM z
    WHERE x * x + y * y < 16::float
      AND i < 27
 )
 SELECT array_to_string(array_agg(substring(' .,,,-----++++%%%%@@@@#### ',
                                  greatest(i,1), 1)),'')
  FROM (
        SELECT ix, iy, max(i) AS i
          FROM z
         GROUP BY iy, ix
         ORDER BY iy, ix
       ) AS zt
 GROUP BY iy
 ORDER BY iy
```

Original idea and T-SQL query by Graeme Job

Not so useful examples...