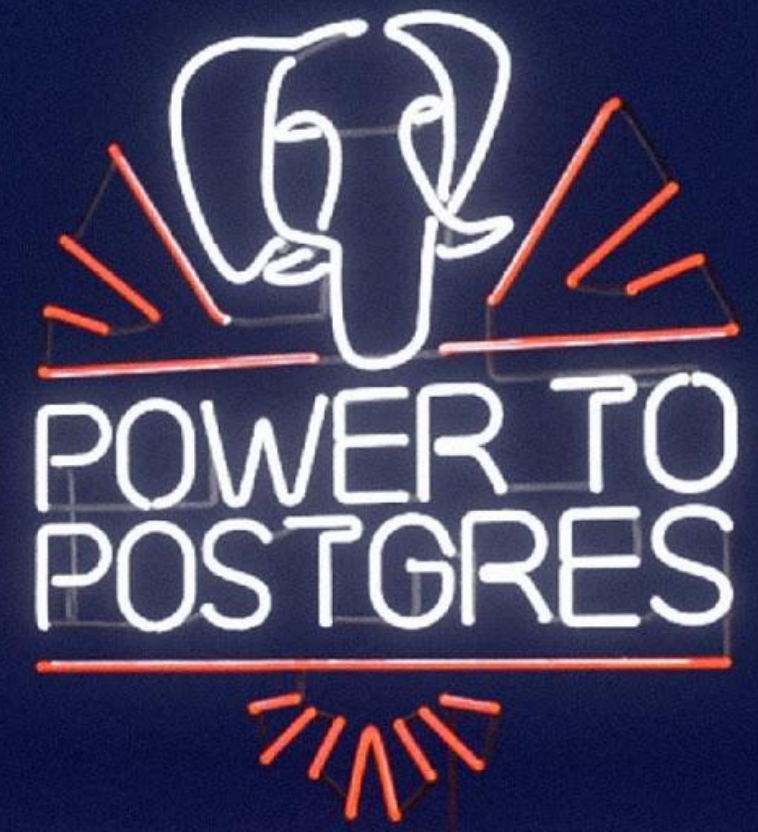
Graph Queries with PostgreSQL

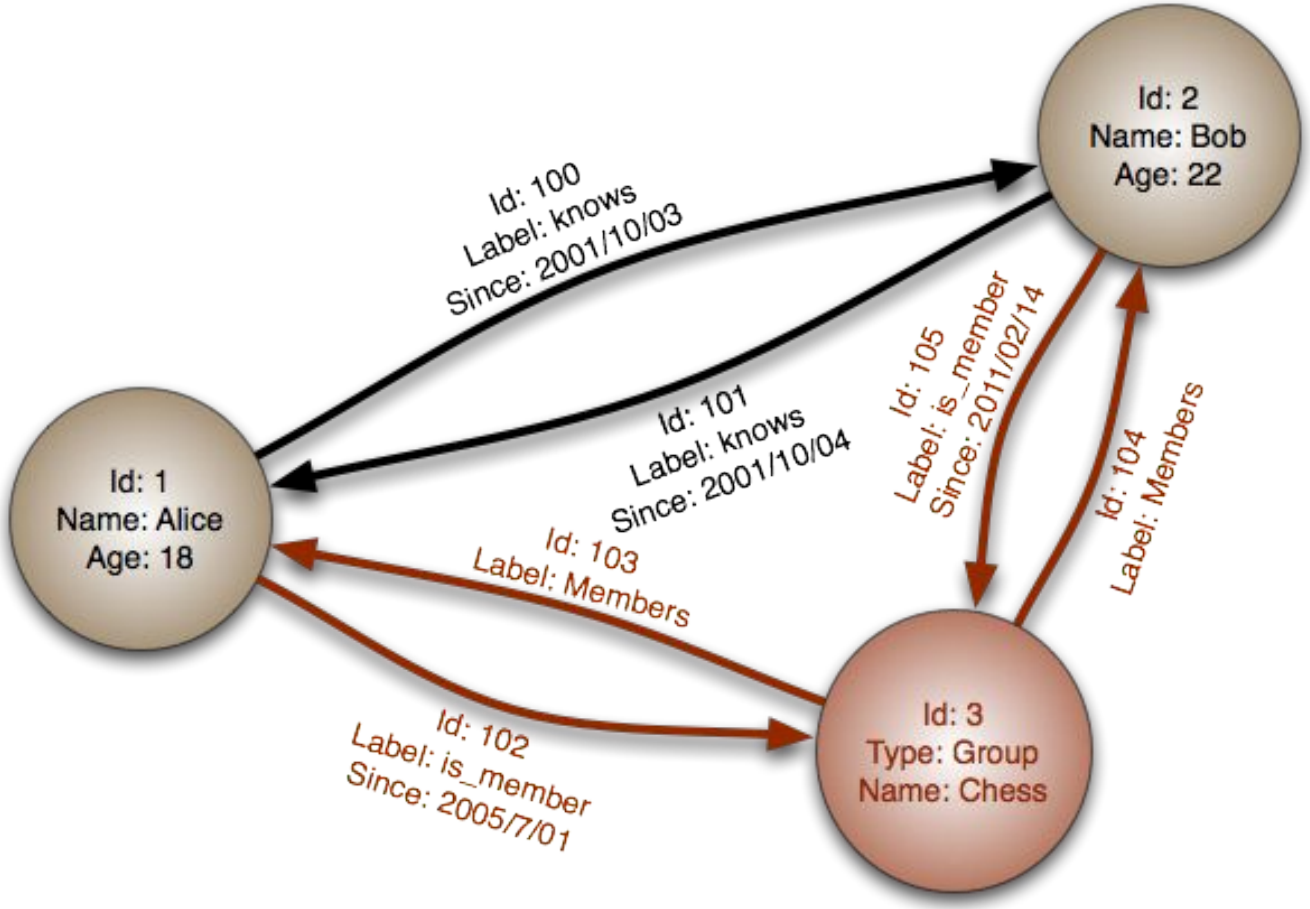
Simon Riggs Postgres Fellow

4Q 2021



EDB

What is a Graph?



NODE
a data item

EDGE
a link between *nodes*, often one direction only

EDB
POWER TO POSTGRES

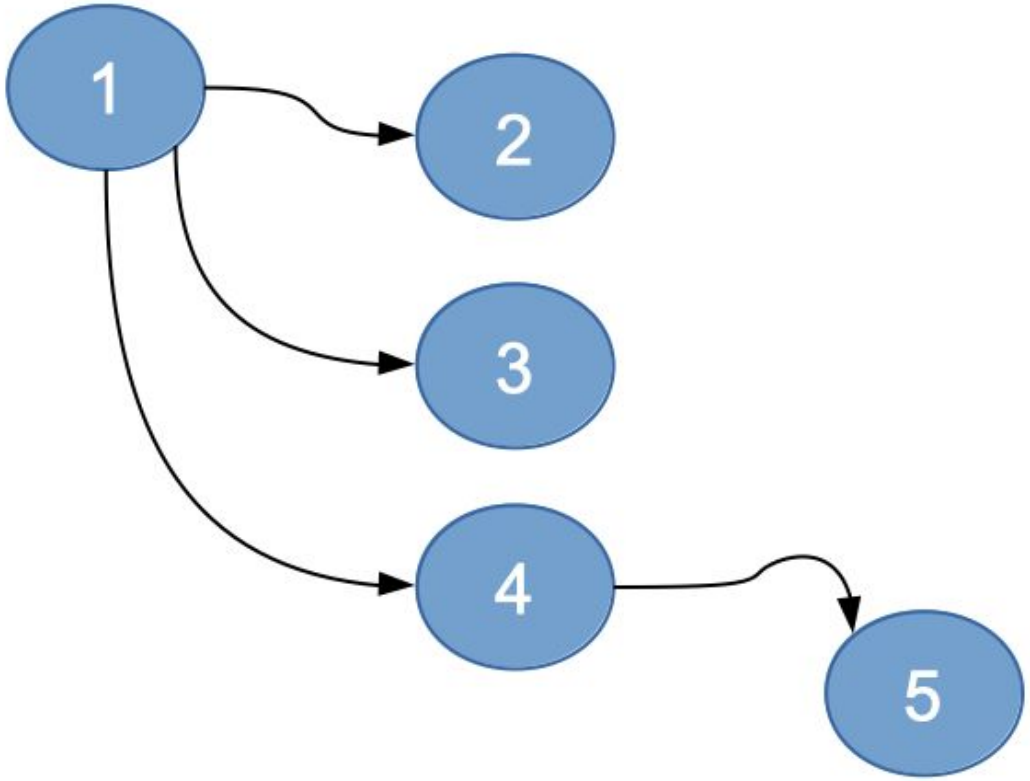
Why should PostgreSQL users care?

- Mixed Models are common
- IATA Data Model
- Bill of Materials (...but never was just a tree)
- Hypertext has multiple directed references (URIs)
- Social Media applications
- Criminal/Fraud analysis

- Some parts of many applications...

- GIS/Mapping applications? No, use pgRouting!

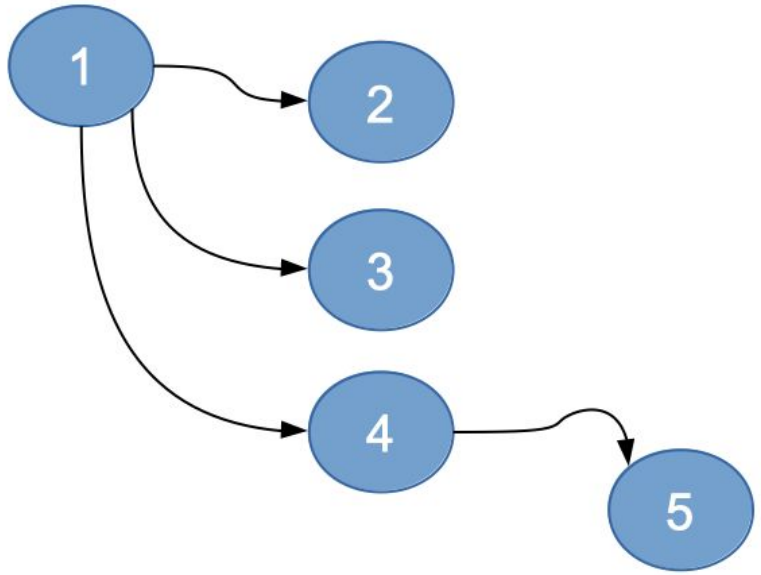
Start with a very simple Graph



Relational Model of Graph

```
CREATE TABLE nodes
(Node INTEGER NOT NULL
,Primary Key (Node));

CREATE TABLE edges
(FromNode INTEGER NOT NULL
   References nodes (Node)
,ToNode     INTEGER NOT NULL
   References nodes (Node)
,Primary Key(FromNode, ToNode));
```

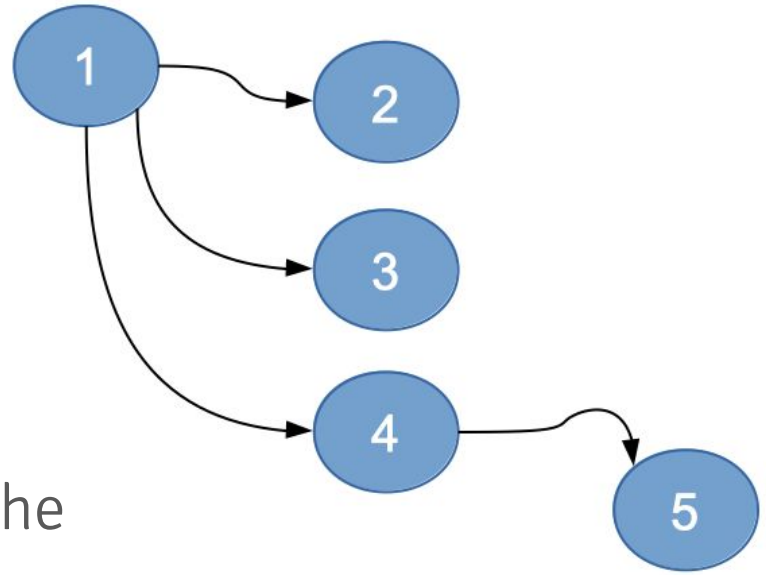


Simple SQL to traverse graph

- ```
  SELECT n2.node
  FROM nodes n1 JOIN edges e
    ON n1.node = e.fromnode
  JOIN nodes n2
    ON e.tonode = n2.node
  WHERE n1.node = ?;
  ```

- Then execute SQL “N” times times to traverse the whole graph

- Seems inelegant….maybe we need…


  ```

...a Query Language other than SQL?

- https://en.wikipedia.org/wiki/Graph_database
  "Retrieving data from a graph database requires a query language other than SQL, which was designed for the manipulation of data in a relational system and therefore cannot “elegantly” handle traversing a graph."

- Proprietary Languages
  - Neo4j - Cypher **=> AgensGraph**
  - TigerGraph - GSQL
  - ArangoDB - AQL
- SPARQL - all of which can be written in SQL
- GQL

Recursive Queries in SQL

- Oracle SQL has supported CONNECT BY … PRIOR syntax for ~30 yrs
- SQL Standard has supported WITH RECURSIVE syntax since SQL:1999
- PostgreSQL has supported SQL Standard syntax for recursive queries since 8.4 on July 1, 2009,
  so fully working and available for >12 years…

Example SQL query WITH RECURSIVE

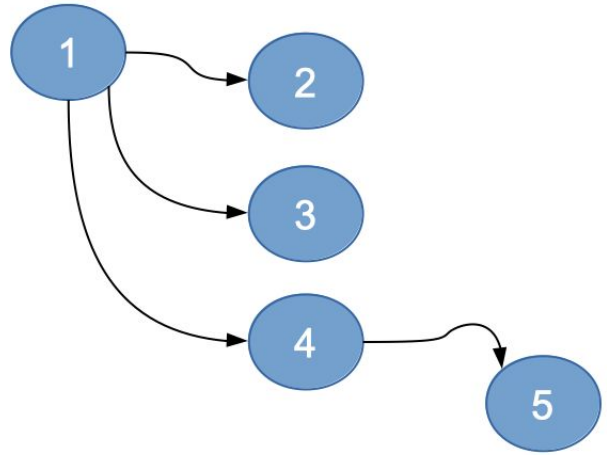
```
with recursive
search_graph(node ,edges ,path)
as (
   select *, ARRAY[g.node]
   from graph g where node = 1
union all
   select g.node ,g.edges
     ,path || g.node
   from graph g, search_graph sg
   where g.node = any(sg.edges)
) select * from search_graph
order by node;
```

Post-relational Graph Model

Using PostgreSQL arrays

```
CREATE TABLE graph
(Node INTEGER NOT NULL
,Edges INTEGER[]
,Primary Key (Node));
```

- Uses single table
- Denormalized with an array of integers
- FKs not required, just optional
- Simplifies indexes and queries
- Use row types for edge array
- Bidirectional graphs need multiple DML
- Don’t try to use CTIDs!



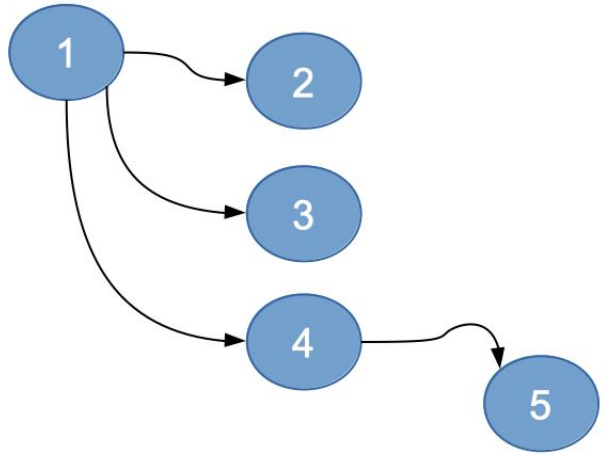

```
SELECT * FROM graph
ORDER BY node;
 node |  edges
------+---------
    1 | {2,3,4}
    2 |
    3 |
    4 | {5}
    5 |
(5 rows)
```

EDB POWER TO POSTGRES

Postgres Graph Query (1)

Recursive SQL Query

```
with recursive
search_graph(node ,edges ,path)
as (
    select node, edges, ARRAY[g.node]
    from graph g where node = 1
union all
    select g.node ,g.edges
      ,path || g.node
    from graph g, search_graph sg
    where g.node = any(sg.edges)
) select * from search_graph
order by node;
```

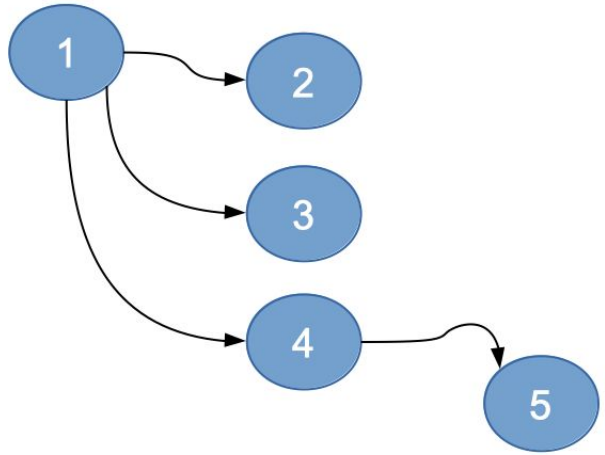



```
SELECT * FROM graph
ORDER BY node;
 node |  edges
------+---------
    1 | {2,3,4}
    2 |
    3 |
    4 | {5}
    5 |
(5 rows)
```

Postgres Graph Query (2)

Defining the initial/non-recursive term

```
with recursive
search_graph(node ,edges ,path)
as (
    select node, edges, ARRAY[g.node]
    from graph g where node = 1
union all
    select g.node ,g.edges
      ,path || g.node
    from graph g, search_graph sg
    where g.node = any(sg.edges)
) select * from search_graph
order by node;
```

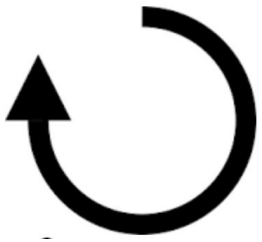


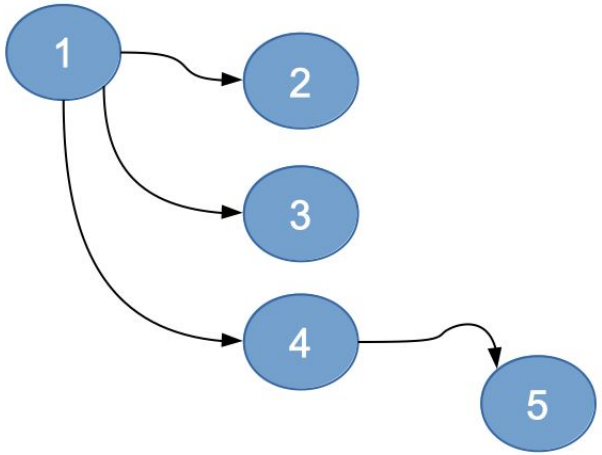

```
SELECT * FROM graph
ORDER BY node;
 node |  edges
------+---------
    1 | {2,3,4}
    2 |
    3 |
    4 | {5}
    5 |
(5 rows)
```

Postgres Graph Query (3)

Defining the recursive term

```
with recursive
search_graph(node ,edges ,path)
as (
    select node, edges, ARRAY[g.node]
    from graph g where node = 1
union all
    select g.node ,g.edges
      ,path || g.node
    from graph g, search_graph sg
    where g.node = any(sg.edges)
) select * from search_graph
order by node;
```






```
SELECT * FROM graph
ORDER BY node;
 node |  edges
------+---------
    1 | {2,3,4}
    2 |
    3 |
    4 | {5}
    5 |
(5 rows)
```

What type of query results do you want?

- See all paths between all nodes? - lots of rows returned!
- See all paths between any two nodes?
  - Need to add a WHERE clause to filter results
  - Helps reduce the size of final results
- See one path, e.g. shortest path between two nodes
  - Add further clauses to remove intermediate results
- More tightly defined queries execute faster (Analytics → OLTP)

Problems in the data?

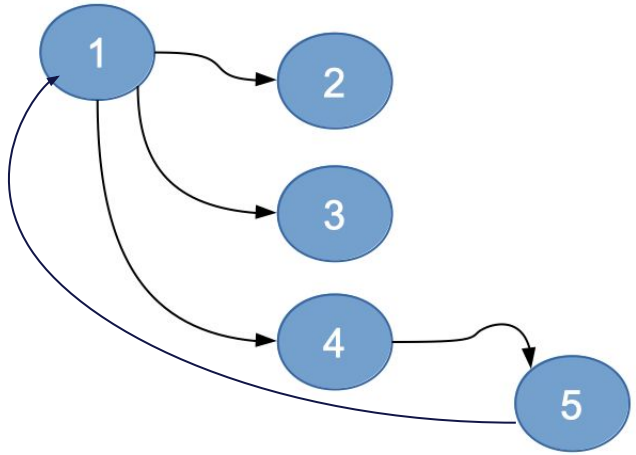
- If the edges cause cycles in the graph, the query and continue forever
- Maybe your data does not contain cycles now, but it might in the future, so you should add non-cycling clauses to the query

- Also possible for there to be multiple paths between two nodes, which can magnify the search time significantly
- May need to add clauses to the query to remove multi-path

Postgres Graph Query (5)

Preventing cycles in the query

```
with recursive
search_graph(node ,edges ,path)
as (
    select node, edges, ARRAY[g.node]
    from graph g where node = 1
union all
    select g.node ,g.edges
      ,path || g.node
    from graph g, search_graph sg
    where g.node = any(sg.edges)
      and g.node <> all(sg.path)
) select * from search_graph
order by node;
```



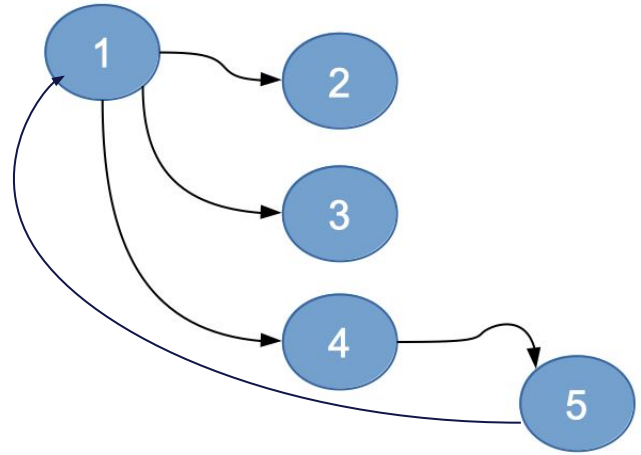

```
SELECT * FROM graph
ORDER BY node;
 node |  edges
------+---------
    1 | {2,3,4}
    2 |
    3 |
    4 | {5}
    5 | {1}   CYCLE!
(5 rows)
```

EDB POWER TO POSTGRES

Postgres Graph Query (6)

Collecting depth/level info during execution

```
with recursive
search_graph(node ,edges ,path, level)
as (
    select node, edges, ARRAY[g.node], 1
    from graph g where node = 1
union all
    select g.node ,g.edges
      ,path || g.node, level+1
    from graph g, search_graph sg
    where g.node = any(sg.edges)
      and g.node <> all(sg.path)
) select * from search_graph
order by node;
```

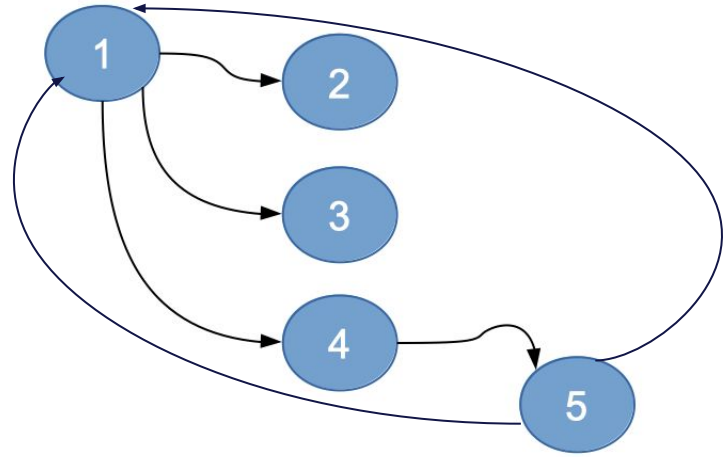



```
SELECT * FROM graph
ORDER BY node;
 node |  edges
------+---------
    1 | {2,3,4}
    2 |
    3 |
    4 | {5}
    5 | {1}
(5 rows)
```

Postgres Graph Query (7)

Removing multiple paths (1 of 2 ways)

```
with recursive
search_graph(node ,edges ,path, level)
as (select node, edges, ARRAY[g.node], 1
     from graph g where node = 1
union all
    select id, edge, path, level from (
    select g.node ,g.edges
      ,path || g.node, level+1,
    row_number() OVER (PARTITION BY sg.id, k.edge ORDER BY sg.level) as rn
    from graph g, search_graph sg
    where g.node = any(sg.edges)
      and g.node <> all(sg.path)
) sg_all where rn = 1
) select * from search_graph
order by node;
```




```
SELECT * FROM graph
ORDER BY node;
 node |  edges
------+---------
    1 | {2,3,4}
    2 |
    3 |
    4 | {5}
    5 | {1}
    5 | {1}
(6 rows)
```

Postgres Graph Query - EXPLAIN

EXPLAIN (COSTS OFF) shows efficient index paths

```
 Sort
   Sort Key: search_graph.node
   CTE search_graph
     ->  Recursive Union
           ->  Index Scan using graph_pkey on graph g
               Index Cond: (node = 1)
           ->  Nested Loop
                 ->  WorkTable Scan on search_graph sg
                 ->  Index Scan using graph_pkey on graph g_1
                     Index Cond: (node = ANY (sg.edges))
                     Filter: (node <> ALL (sg.path))
   ->  CTE Scan on search_graph
(12 rows)
```

EDB POWER TO POSTGRES

Postgres Graph Query (Full example)

Shortest path query, avoiding loops, removing multipaths

```
with recursive
search_graph(id, edge, path, level) as (
select k_person1id, edge, ARRAY[k_person1id], 1
 from knows2 k
 where k_person1id = :person1Id::bigint
union all
select id, edge, path, level from (
 select sg.id as id, k.edge, path || k.k_person1id as path, sg.level+1 as level
       ,row_number() OVER (PARTITION BY sg.id, k.edge ORDER BY sg.level) as rn
  from knows2 k, search_graph sg
  where k.k_person1id = any(sg.edge)
    and k.k_person1id <> all(path)
 ) sg_all where rn = 1
)
select level from search_graph
where :person2Id::bigint = any(edge)
limit 1;
```

Postgres Graph Query (EXPLAIN ANALYZE)

PG14 raw plan

```
                                                QUERY PLAN
-----------------------------------------------------------------------------------------------------------------------------------------------------
 Limit  (cost=20282226.50..20282231.00 rows=1 width=4) (actual time=1774.962..1774.965 rows=1 loops=1)
   CTE search_graph
     ->  Recursive Union  (cost=0.42..20282226.50 rows=391385 width=52) (actual time=0.012..1774.861 rows=221 loops=1)
           ->  Index Only Scan using knows_pkey on knows k  (cost=0.42..31.74 rows=875 width=52) (actual time=0.011..0.014 rows=10 loops=1)
                 Index Cond: (k_person1id = '1'::bigint)
                 Heap Fetches: 0
           ->  Subquery Scan on sg_all  (cost=1715031.75..2027436.71 rows=39051 width=52) (actual time=441.342..443.690 rows=53 loops=4)
                 Filter: (sg_all.rn = 1)
                 Rows Removed by Filter: 2948
                 ->  WindowAgg  (cost=1715031.75..1929810.16 rows=7810124 width=64) (actual time=441.340..443.485 rows=3000 loops=4)
                       ->  Sort  (cost=1715031.75..1734557.06 rows=7810124 width=60) (actual time=441.242..441.466 rows=3002 loops=4)
                             Sort Key: sg.id, k_1.k_person2id, sg.level
                             Sort Method: quicksort  Memory: 26kB
                             ->  Hash Join  (cost=33249.47..233607.06 rows=7810124 width=60) (actual time=268.630..439.844 rows=3002 loops=4)
                                   Hash Cond: (sg.edge = k_1.k_person1id)
                                   Join Filter: (k_1.k_person1id <> ALL (sg.path))
                                   ->  WorkTable Scan on search_graph sg  (cost=0.00..175.00 rows=8750 width=52) (actual time=0.001..0.022 rows=55 loops=4)
                                   ->  Hash  (cost=15622.21..15622.21 rows=1014021 width=16) (actual time=268.322..268.322 rows=1014021 loops=4)
                                         Buckets: 131072 (originally 131072)  Batches: 32 (originally 16)  Memory Usage: 3556kB
                                         ->  Seq Scan on knows k_1  (cost=0.00..15622.21 rows=1014021 width=16) (actual time=0.006..105.387 rows=1014021 loops=4)
   ->  CTE Scan on search_graph  (cost=0.00..8806.16 rows=1957 width=4) (actual time=1774.961..1774.961 rows=1 loops=1)
         Filter: (edge = '100000'::bigint)
         Rows Removed by Filter: 220
 Planning Time: 0.244 ms
 Execution Time: 1775.501 ms
```

Postgres Graph Query (EXPLAIN ANALYZE)

PG14 plus reduced estimate

```
                                         QUERY PLAN
---------------------------------------------------------------------------------------------------------------------------------------------------------
 Limit  (cost=2080213.70..2080218.20 rows=1 width=4) (actual time=17.119..17.121 rows=1 loops=1)
   CTE search_graph
     ->  Recursive Union  (cost=0.42..2080213.70 rows=39925 width=52) (actual time=0.014..17.035 rows=221 loops=1)
           ->  Index Only Scan using knows_pkey on knows k  (cost=0.42..31.74 rows=875 width=52) (actual time=0.013..0.016 rows=10 loops=1)
                 Index Cond: (k_person1id = '1'::bigint)
                 Heap Fetches: 0
           ->  Subquery Scan on sg_all  (cost=176697.87..207938.35 rows=3905 width=52) (actual time=1.902..4.236 rows=53 loops=4)
                 Filter: (sg_all.rn = 1)
                 Rows Removed by Filter: 2948
                 ->  WindowAgg  (cost=176697.87..198175.70 rows=781012 width=64) (actual time=1.901..4.033 rows=3000 loops=4)
                       ->  Sort  (cost=176697.87..178650.40 rows=781012 width=60) (actual time=1.801..2.020 rows=3002 loops=4)
                             Sort Key: sg.id, k_1.k_person2id, sg.level
                             Sort Method: quicksort  Memory: 26kB
                             ->  Nested Loop  (cost=0.42..41526.38 rows=781012 width=60) (actual time=0.007..0.931 rows=3002 loops=4)
                                   ->  WorkTable Scan on search_graph sg  (cost=0.00..17.50 rows=875 width=52) (actual time=0.000..0.005 rows=55 loops=4)
                                   ->  Index Only Scan using knows_pkey on knows k_1  (cost=0.42..38.51 rows=893 width=16) (actual time=0.003..0.010 rows=55 loops=220)
                                         Index Cond: (k_person1id = sg.edge)
                                         Filter: (k_person1id <> ALL (sg.path))
                                         Heap Fetches: 0
   ->  CTE Scan on search_graph  (cost=0.00..898.31 rows=200 width=4) (actual time=17.119..17.119 rows=1 loops=1)
         Filter: (edge = '100000'::bigint)
         Rows Removed by Filter: 220
 Planning Time: 0.173 ms
 Execution Time: 17.198 ms
```

x100 faster!

EDB POWER TO POSTGRES

PostgreSQL Graph Query Performance

- Linked Data Benchmark Council (LDBC)
  - Same team who published TPC-H analyses

- Social Network Benchmark (SNB)
- All queries can be expressed in SQL for PostgreSQL
- Graph OLTP Benchmark
  - PostgreSQL beats Neo4j Community Edition

- Graph BI Benchmark

Other PostgreSQL Performance Results

- Alibaba Cloud published details of a PostgreSQL graph benchmark at 12,000TPS/2.1ms per query with 5 billion nodes
- https://www.arangodb.com/2015/10/benchmark-postgresql-mongodb-arangodb/
- “Database Performance Comparisons: An Inspection of Fairness”, Uwe Hohenstein and Martin Jergler, 2019
  - “Moreover, we refute some stated issues about the bad performance of relational systems by using a PostgreSQL database for commonly used test scenarios.”
- Various papers on PostgreSQL on LDBC Benchmarks

PostgreSQL 14 features

- SQL Standard features added for recursive (graph) queries
- [SEARCH (BREADTH | DEPTH) FIRST BY col, …
  SET search_seq_col_name]
- [CYCLE col, … SET cycle_mark_col_name
  [TO cycle_mark_value DEFAULT cycle_mark_default]
  USING cycle_path_col_name]

Postgres Graph Query (New PG14)

Shortest path query, avoiding loops, removing multipaths

```
with recursive
search_graph(id, edge, level)
as (
select k_person1id, k_person2id, 1
 from knows k
 where k_person1id = :person1Id::bigint
union all
select sg.id, k.k_person2id, sg.level+1
 from knows k, search_graph sg
 where k.k_person1id = sg.edge
) cycle id, edge set is_cycle using path
select level from search_graph
where edge = :person2Id::bigint
limit 1;
```

EDB POWER TO POSTGRES

PostgreSQL Usage Simplification

- ```
  CREATE VIEW <table name> AS
  WITH RECURSIVE <table name> (<view column list>)

  AS (<query expression>)

  SELECT <view column list> FROM <table name>
  ```

- Encapsulates complexity, so many developers can use recursive queries easily without needing to understand them
  ```

PostgreSQL Recursive DML

- Execute recursive DML for
  - UPDATE
  - INSERT
  - DELETE

- Feature is unique extension in SQL for PostgreSQL

PostgreSQL Multi-Model DBMS

- Everything discussed can work with any data type
  - Normal columns: Integers/BigInt, Text/URLs
  - Row Types
  - Document types: JSON/JSONB, XML

- Graph Schema can be easily and significantly customized by you to include your additional requirements, without affecting **performance**

SQL Standard Roadmap

- SQL:202(n) will contain SQL/PGQ
  - PGQ=Property Graph Query

- New graph query language GQL, separate from SQL

SQL/PGQ Example

Native graph query syntax in SQL

```
select start_id, end_id
from graph graph_table (
  match (n1:node)-[:edge{1,5}]->(n2:node)
  columns n1.id as start_id, n2.id as end_id
)
order by start_id;
```

Poor Market Understanding?

- https://stackoverflow.com/questions/20776718/best-way-to-model-graph-data-in-postgresql
  "I realize dedicated graph databases like [GraphDB] are best suited for this, ..."
  Closed. This question is opinion-based.
- https://www.reddit.com/r/PostgreSQL/comments/8mdsxr/does_postgresql_11_support_graph_database/

  "I do not see any support for graph database in PostgreSQL 11 Beta 1. I thought it was planned on the roadmap."
- Joe Celko's book "Trees & Hierarchies in SQL for Smarties" (2012) does cover graphs, but not detailed enough

PostgreSQL

- Can one DBMS be best at everything? **No**
- Can a DBMS with huge numbers of contributors collect together to produce something no one mind could contemplate, covering multiple use cases?
  Yes, but slowly
- Can one DBMS provide the facilities for multiple additional features via extensibility? **Definitely**

EDB supercharges Postgres

Products, services, and support for teams who need to do more and go faster.

Databases

PostgreSQL and extensions for enterprise workloads

Tools

Monitoring, management, scalability, high availability

Deployments

On-prem to the cloud, virtual machines to Kubernetes

Expertise

24/7 technical support, remote DBAs, professional services

EDB POWER TO POSTGRES

We're the PostgreSQL experts

1986
The design of PostgreSQL

1996
Birth of PostgreSQL

2004
EDB is founded

2007
2ndQuadrant launched

2020
EDB acquires 2ndQuadrant

EDB TEAM INCLUDES:

- 300+ PostgreSQL technologists
- 26 PostgreSQL community contributors and committers
- Including founders and leaders in PostgreSQL Community

Key PostgreSQL Contributions

EDB
- Heap Only Tuples (HOT)
- Materialized Views
- Parallel Query
- JIT Compilation
- Serializable Parallel Query

2ndQuadrant
- Hot Standby
- Logical Replication
- Transaction Control
- Generated Columns

No company has contributed more to PostgreSQL

The most PostgreSQL experts

EDB team includes:

300+ PostgreSQL technologists

26 PostgreSQL community contributors and committers

Including founders and leaders like

Michael Stonebraker
"Father of Postgres"
and EDB Advisor

Bruce Momjian
Co-founder, PostgreSQL
Development Corp and
PostgreSQL Core Team

Peter Eisentraut
PostgreSQL
Core Team member

Robert Haas
PostgreSQL Major
Contributor and Committer

Simon Riggs
PostgreSQL Major
Contributor, Founder
of 2ndQuadrant

EDB supported databases

PostgreSQL

Open source PostgreSQL

- EDB continues to be committed to advancing features in collaboration with the broader community

EDB Postgres Extended

EDB proprietary distribution

- SQL compatible with PostgreSQL, extended for stringent availability and advanced replication needs
- Formerly known as 2ndQPostgres

EDB Postgres Advanced

EDB proprietary distribution

- SQL compatible with Oracle, reduces effort to migrate applications and data to PostgreSQL
- Additional value-add enterprise features

EDB POWER TO POSTGRES

Thank you