

RELATIONAL OR GRAPHS? WHY NOT BOTH?

ANTS AASMA

pgconf.eu 2026

Hi



About me

- Lead database consultant at Cybertec
- Performance, HA, development



Chapter 1: Isomorphism



Relational databases and graph databases

Relational



Graph



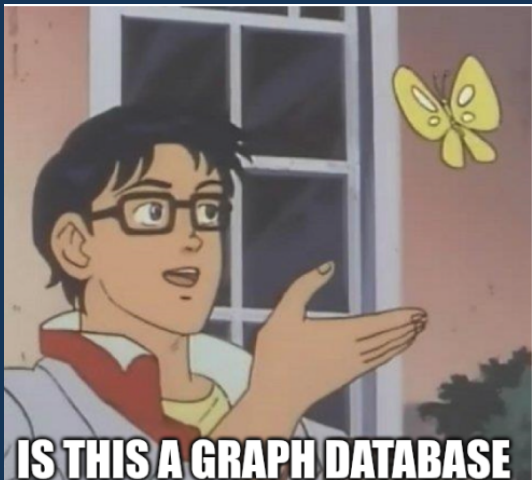
All data is relational



All data is relational

All data is graphs





```
CREATE TABLE vertices (  
    id bigserial PRIMARY KEY,  
    kind text,  
    attributes jsonb  
);  
  
CREATE TABLE edges (  
    src int8 REFERENCES vertices(id),  
    dst int8 REFERENCES vertices(id),  
    kind text,  
    attributes jsonb,  
    PRIMARY KEY (src, dst, kind)  
);
```



It's the programming model

The difference is the programming model / query API



It's the programming model

The difference is the programming model / query API

Memory stores bytes, disks store blocks, CPU calculates numbers.



It's the programming model

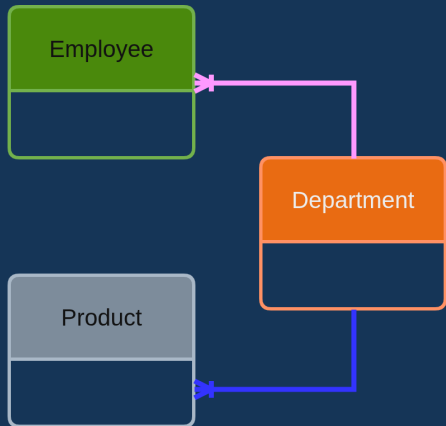
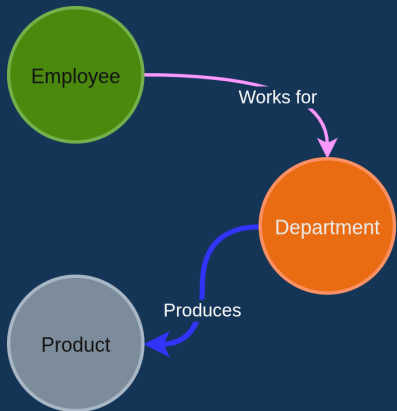
The difference is the programming model / query API

Memory stores bytes, disks store blocks, CPU calculates numbers.

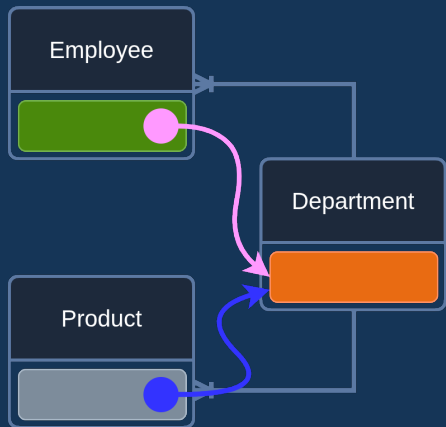
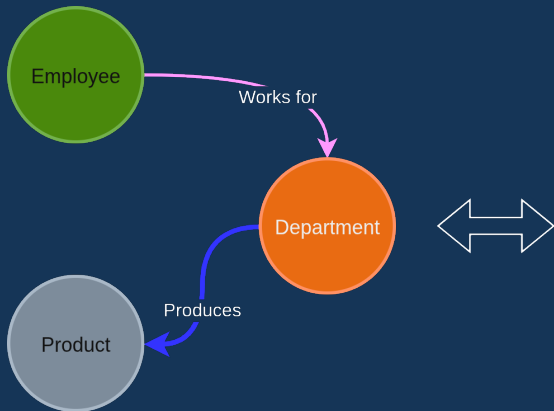
Database maps our schema, data and queries to algorithms and data structures.



Schema is not the graph



It's all rows



Cypher $\Leftrightarrow$ SQL



Navigating edges is joins

```
MATCH (:Person {name:'Tom Hanks'})-[r:ACTED_IN]->(m:Movie) RETURN *
```

```
SELECT *  
FROM Person  
  JOIN Roles ON Person.id = Roles.person_id  
  JOIN Movie ON Roles.movie_id = Movie.id  
WHERE Person.name = 'Tom Hanks'
```



Variable length paths

```
MATCH p = ALL SHORTEST (p1:Person)-[:Friend]-+(p2:Person)
WHERE p1.name = "John Doe" AND p2.name = "Jane Doe"
RETURN [n in nodes(p) | n.name] AS friends
```

SQL: ...



Actual SQL

```
WITH RECURSIVE shortest_path AS (  
  SELECT p.id AS current_id, ARRAY[p.name]::text[] AS friends_path, 1 AS path_length  
  FROM person p WHERE p.name = 'John Doe'  
  UNION ALL  
  -- Recursive member: Traverse friends while tracking the path  
  SELECT f.friend_id, sp.friends_path || p2.name, sp.path_length + 1  
  FROM shortest_path sp  
    JOIN friends_edge f ON sp.current_id = f.person_id  
    JOIN person p2 ON f.friend_id = p2.id  
  WHERE NOT (p2.name = ANY(sp.friends_path))  
    AND NOT ('Jane Doe' = ANY(sp.friends_path))  
)  
target_paths AS (  
  SELECT friends_path, path_length FROM shortest_path  
  WHERE friends_path[array_length(friends_path, 1)] = 'Jane Doe'  
)  
-- Return only the paths that match the minimum length (ALL SHORTEST)  
SELECT friends_path AS friends FROM target_paths  
WHERE path_length = (SELECT MIN(path_length) FROM target_paths);
```



Graph databases and relational databases can solve the same problems.

This does not mean that the solutions are equally simple.

The real difference is the language we express ourselves in.



Chapter 2: SQL/PGQ



Standardize the isomorphism

```
CREATE PROPERTY GRAPH social
VERTEX TABLES (
    Person
)
EDGE TABLES (
    Friends_edge
        SOURCE KEY (person_id) REFERENCES Person (id)
        DESTINATION KEY (friend_id) REFERENCES Person (id)
        LABEL Friend
);
```



Querying

```
SELECT * FROM GRAPH_TABLE(social
  MATCH (p1:Person WHERE p1.name = 'John Doe')->[f:Friend]->(p2:Person)
  COLUMNS (p2.name)
)
```



Shortest path

```
SELECT * FROM GRAPH_TABLE(social
  MATCH p = ANY SHORTEST (p1:Person WHERE p1.name = 'John Doe')
    ->[f:Friend]->+(p2:Person WHERE p2.name = 'Jane Doe')
  COLUMNS (path_length(p), p2.name))
```



What's not in PG19

- First iteration supported only fixed length paths.
- Implemented in the rewriter.
- SQL/PGQ will return in 20 (fingers crossed)
- Variable length paths in the longer term future.



Chapter 3: Living with the tools we have



Recursive queries

WITH RECURSIVE can be used to implement iteration.

Problems:

- Very verbose to write.
- Executes breadth first.
- Kind of slow.



Understanding recursive query

```
WITH RECURSIVE search_graph(id, link, data, depth) AS (  
    /* Starting point */  
    SELECT g.id, g.link, g.data, 1  
    FROM graph g  
    UNION ALL  
    /* Next step */  
    SELECT g.id, g.link, g.data, sg.depth + 1  
    FROM graph g, search_graph sg  
    WHERE g.id = sg.link  
) CYCLE id SET is_cycle USING path  
SELECT * FROM search_graph;
```



What to optimize for

Graph traversal queries:

- Many dependent steps.
- Large intermediate state.
- Execution time decision for order of steps.



What we need for performance

- Data locality
- Fast vertex sets operations
- Compact representation of large sets
- No network hops in inner loops



How to get data locality, Option A

```
CREATE INDEX ON Friends_edge (person_id, friend_id);
```

- Duplicate data
- Visibility checks can kill index only scans
- 28 bytes per edge
- + Can include arbitrary attributes



How to get data locality, Option B

```
CREATE TABLE friendships (  
    person_id int PRIMARY KEY,  
    friend_ids int[]  
);
```

- 4 bytes per edge asymptotically (24 byte per row overhead)
- Modifications cause write amplification
- Storing extra attributes is tricky.
- SQL/PGQ will not know what to do with it.



Do I smell TOAST?

- Large values (2kB+) get stored out of line.
- Extra index lookup.
- CPU overhead to reassemble the value.
- Increase `toast_tuple_target` to 8160
- Use lz4 compression, or disable compression (`SET STORAGE EXTERNAL`)



Accelerating the set operations

- Array operations are still expensive.
- `pg_roaringbitmap` comes to the rescue.
 - Datatype to store integer sets.
 - Peak memory usage is 1bit per vertex in the graph.
 - 128M rows -> 16MB.
 - SIMD optimized operations



No network hops

- Index lookup takes a couple microseconds.
- Network latency is 100x bigger.
- CPU overhead is 10x bigger.
- Process at least the core loop all inside the database server.



Running on the database

- plpgsql can be used to stitch together the execution steps.
plpgsql : postgres executor <-> Python : Numpy



Overview

- Store data in adjacency list format if feasible.
- Optimize for big sets.
- Implement the algorithm in the database.



Chapter 4: An example



ALL SHORTEST

- Lets use roaring bitmaps to build a shortest path algorithm.
- Bidirectional breadth first search.
- Set based.
- Instead of storing distance on every node, store set of nodes for every distance.



Overview of algorithm

Start with set of start nodes $\{\text{start}\}$ and end nodes $\{\text{end}\}$.



Overview of algorithm

Start with set of start nodes {start} and end nodes {end}.

Find all nodes reachable from {start} that we have not visited yet.

```
SELECT rb_or_agg(out) - visited FROM adjacency
WHERE id = ANY (rb_to_array(:start))
```



Overview of algorithm

Start with set of start nodes {start} and end nodes {end}.

Find all nodes reachable from {start} that we have not visited yet.

```
SELECT rb_or_agg(out) - visited FROM adjacency
WHERE id = ANY (rb_to_array(:start))
```

If this set overlaps with destination we have found a path.



Overview of algorithm

Start with set of start nodes {start} and end nodes {end}.

Find all nodes reachable from {start} that we have not visited yet.

```
SELECT rb_or_agg(out) - visited FROM adjacency
WHERE id = ANY (rb_to_array(:start))
```

If this set overlaps with destination we have found a path.

Otherwise, find shortest path from nodes 1 step away from start.



Overview of algorithm

Start with set of start nodes {start} and end nodes {end}.

Find all nodes reachable from {start} that we have not visited yet.

```
SELECT rb_or_agg(out) - visited FROM adjacency
WHERE id = ANY (rb_to_array(:start))
```

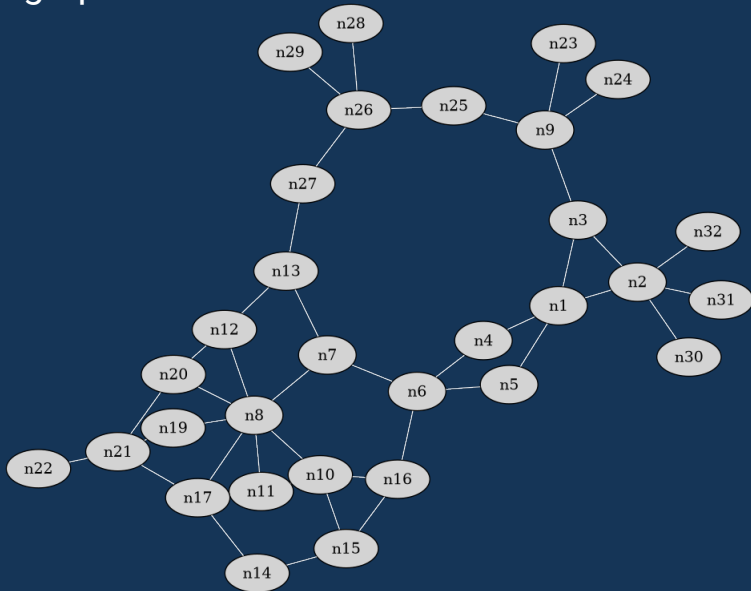
If this set overlaps with destination we have found a path.

Otherwise, find shortest path from nodes 1 step away from start.

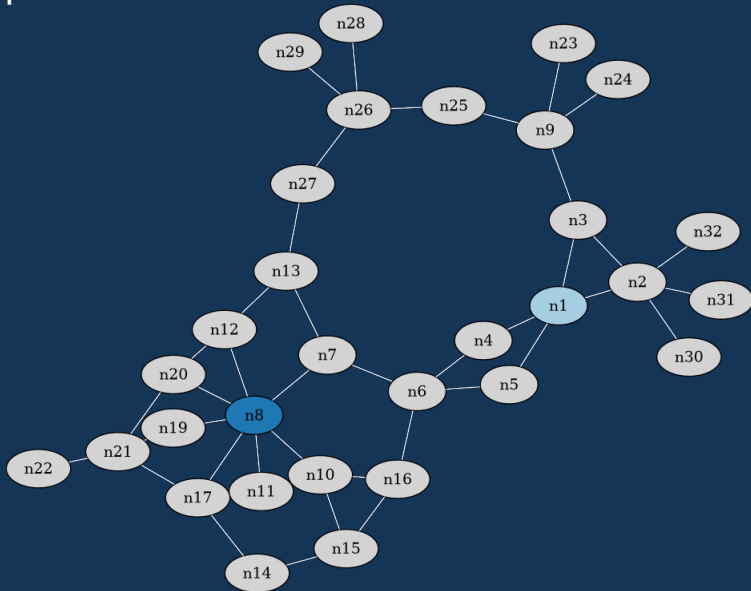
If destination set is smaller expand search from the smaller end.



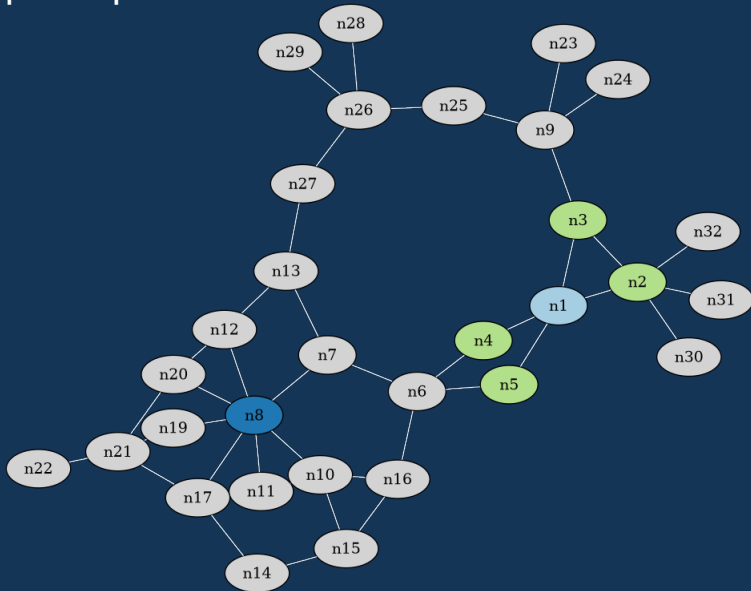
The graph



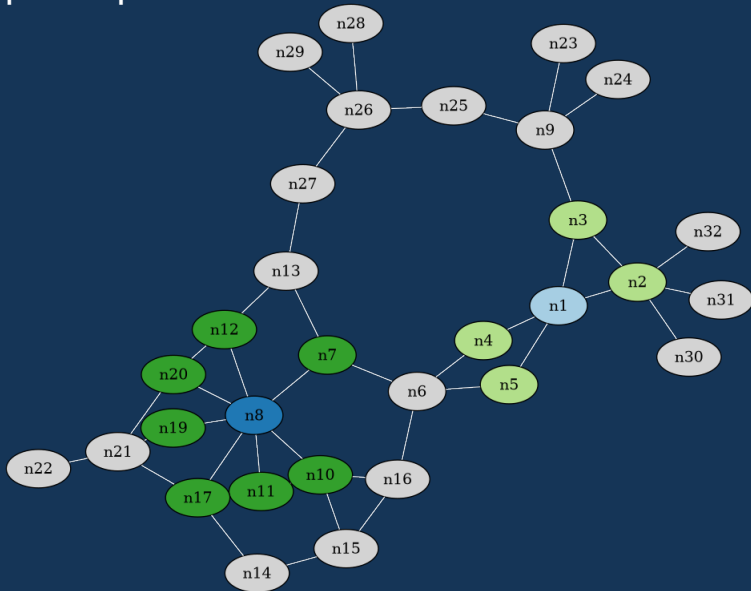
Step 0: start and end



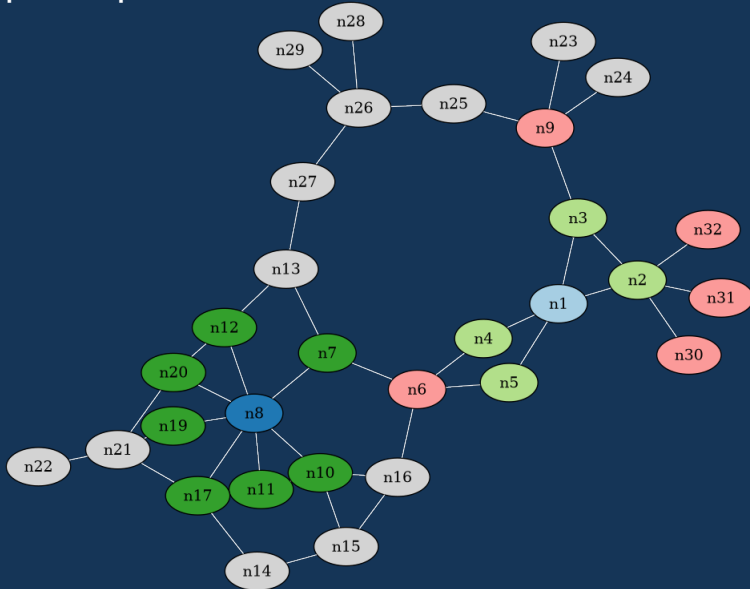
Step 1: expand from the start



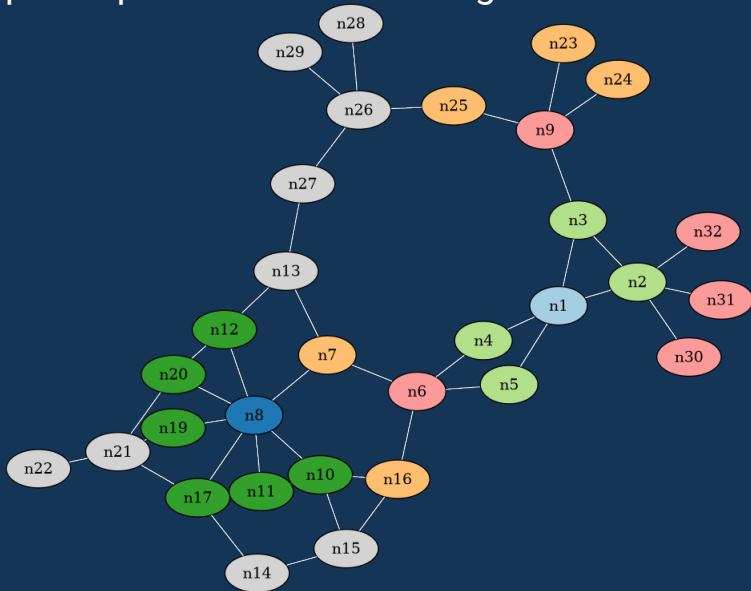
Step 3: expand from the end



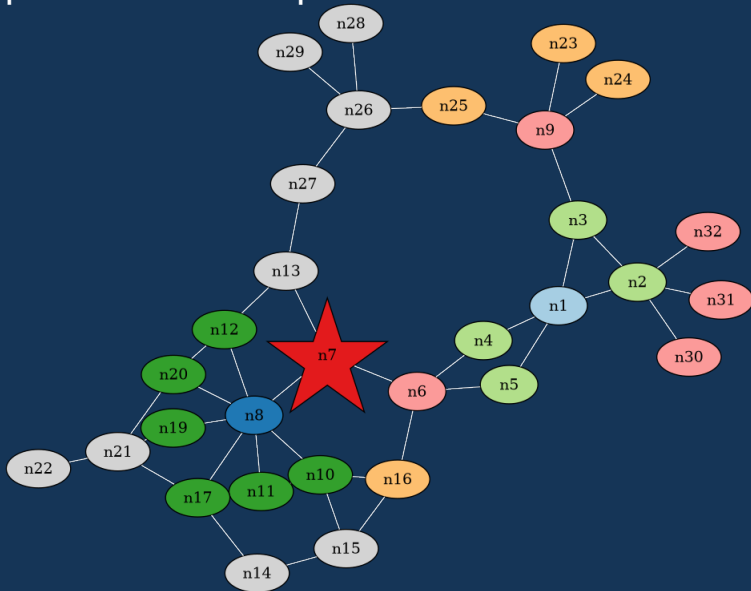
Step 3: expand from the start



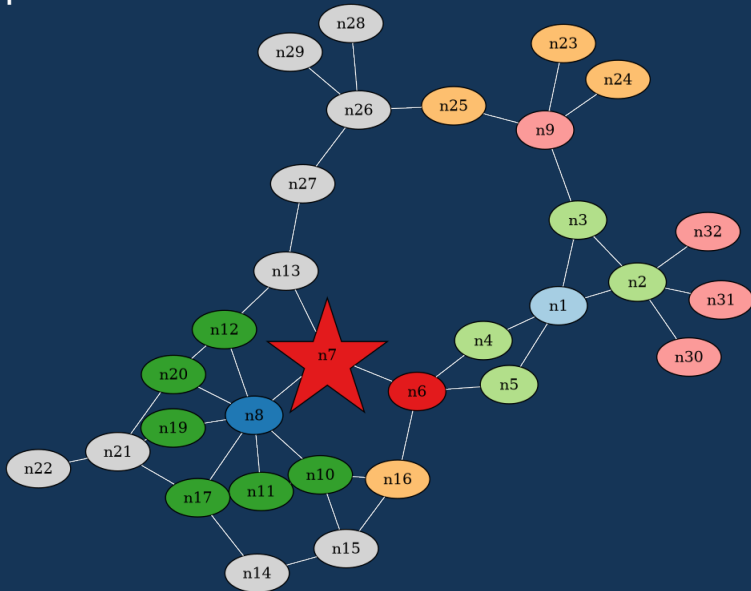
Step 4: expand from the start again



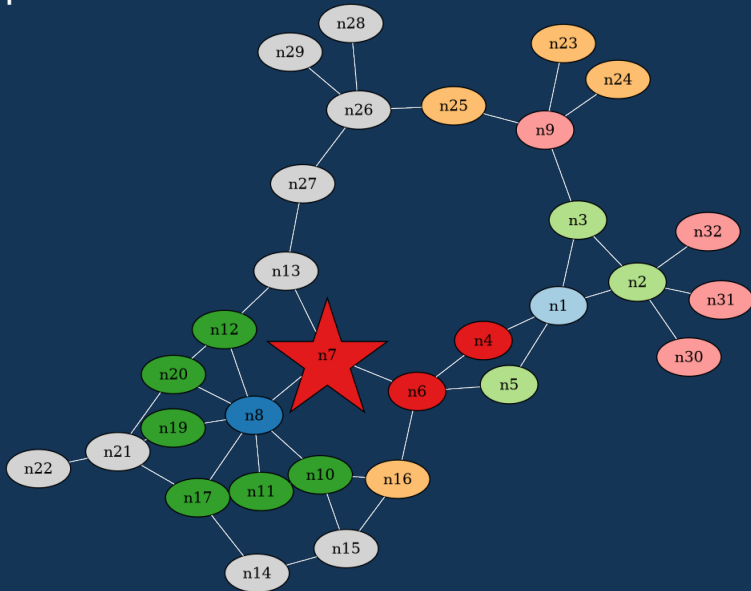
Step 5: Found overlap



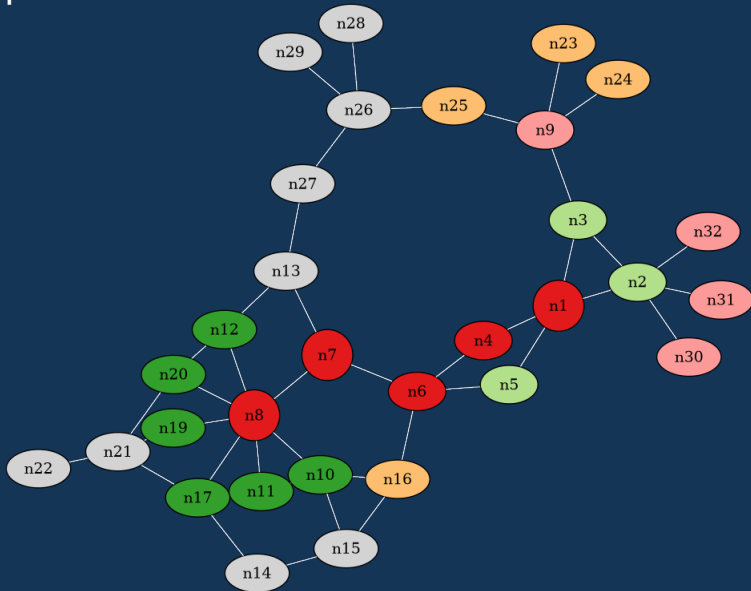
Step 6: Search backwards



Step 6: Search backwards



Step 6: Search backwards



Code time!



Definition

Find all shortest paths between all sources and destinations.

```
CREATE FUNCTION wg_shortest_path(srcs roaringbitmap,  
                                dsts roaringbitmap,  
                                visited roaringbitmap DEFAULT '{}',  
                                forward boolean DEFAULT true)  
  
    RETURNS SETOF integer[]  
    LANGUAGE plpgsql  
    AS $function$  
    DECLARE  
        reachable roaringbitmap;  
        reachable_path int[];  
        prefix int;
```



Termination

If sources and destinations overlap then each overlap is a one element shortest path.

```
BEGIN
  -- No path found
  IF rb_is_empty(srcs) OR srcs IS NULL THEN RETURN; END IF;

  IF srcs && dsts THEN -- Found a path
    FOREACH prefix IN ARRAY rb_to_array(srcs & dsts) LOOP
      RETURN NEXT array[prefix];
    END LOOP;
    RETURN;
  END IF;
```



Bidirectionality

If we have more sources and destinations solve the reverse problem.

```
IF rb_cardinality(srcs) > rb_cardinality(dsts) THEN
    RETURN QUERY SELECT array_reverse(result)
        FROM wg_shortest_path(dsts, srcs, visited, NOT forward) result;
RETURN;
END IF;
```



Take one step

Find reachable nodes not seen yet from the appropriate adjacency list.

```
IF forward THEN
    SELECT rb_or_agg(out) - visited INTO reachable
    FROM webgraph WHERE id = ANY (rb_to_array(sracs));
ELSE
    SELECT rb_or_agg(out) - visited INTO reachable
    FROM webgraph_inv g WHERE id = ANY (rb_to_array(sracs));
END IF;
```



Recursion

Find shortest path between reachable nodes and destination.

If no new sources were found the search terminates here.

```
FOR reachable_path IN SELECT wg_shortest_path(reachable,  
                                              dsts,  
                                              visited | srcs,  
                                              forward)  
  
LOOP
```



Path reassembly

For each found path from reachable nodes find which sources can reach each path.

```
IF forward THEN
  FOR prefix IN SELECT rb_iterate(out & srcs) FROM webgraph_inv
    WHERE id = reachable_path[1] LOOP
    RETURN NEXT prefix || reachable_path;
  END LOOP;
ELSE
  FOR prefix IN SELECT rb_iterate(out & srcs) FROM webgraph
    WHERE id = reachable_path[1] LOOP
    RETURN NEXT prefix || reachable_path;
  END LOOP;
END IF;
```



Finito

```
END LOOP;  
RETURN;  
END; $function$
```



Usage

```
graph=# SELECT * FROM wg_shortest_path('{1138993222}', '{305817984}');  
          wg_shortest_path
```

```
-----  
{1138993222,1138991953,-2056136710,217904522,1785373125,305817948,305817984}  
{1138993222,1138991953,-761419368,410760422,1785373125,305817948,305817984}
```



But is it web scale?



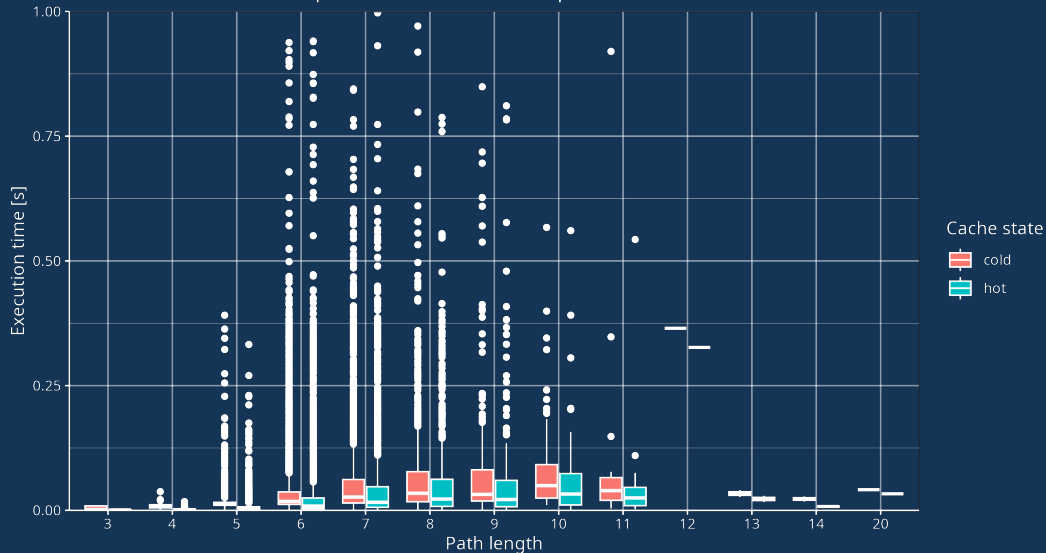
Common crawl hosts graph

- Graph of hyperlinks between all web domains
- commoncrawl.org
- Nodes: 239 812 465
- Edges: 2 730 671 829
- Benchmark: find all shortest paths between two randomly selected nodes



Common crawl hosts graph results

Time to find all shortest paths between random pairs



Not big enough?



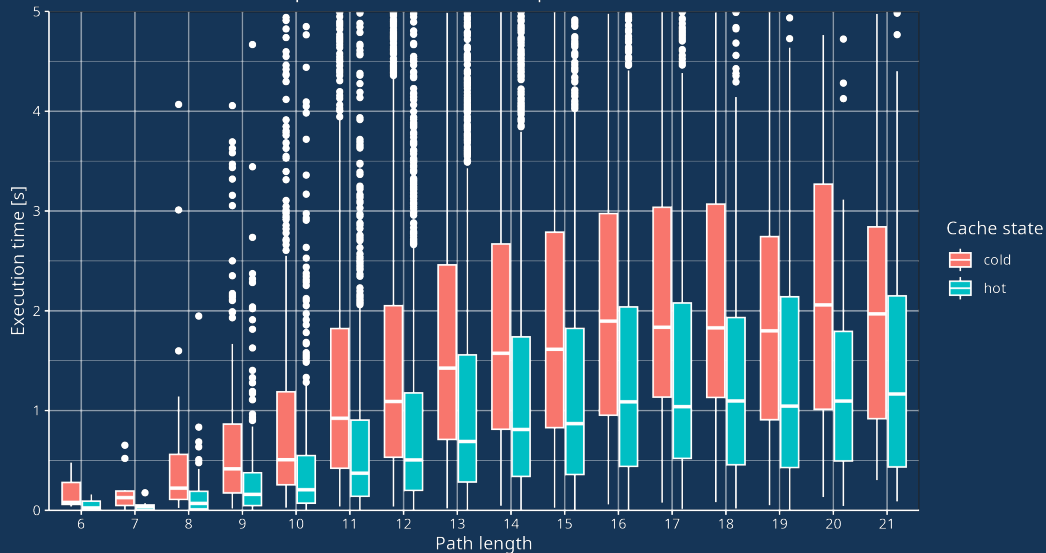
Common crawl full graph

- Graph of hyperlinks between all web **pages**, from 2012
- Nodes: 3 443 082 324
- Edges: 128 736 914 167
- Benchmark: find all shortest paths between two randomly selected nodes
- Size:
 - Forward adjacency list: table: 467 GB pkey index: 122 GB
 - Reverse adjacency list: table: 228 GB pkey index: 79 GB
 - Total: 895GB



Common crawl web graph results

Time to find all shortest paths between random pairs



Chapter 4: What we wont have any time soon



Deferred visibility checks

- Index lookups need to do visibility checks from table data
- Loss of data locality
- Option to join first and run visibility checks last would help



Dynamic join order

- Choose traversal/join order based on actual in and out degrees.

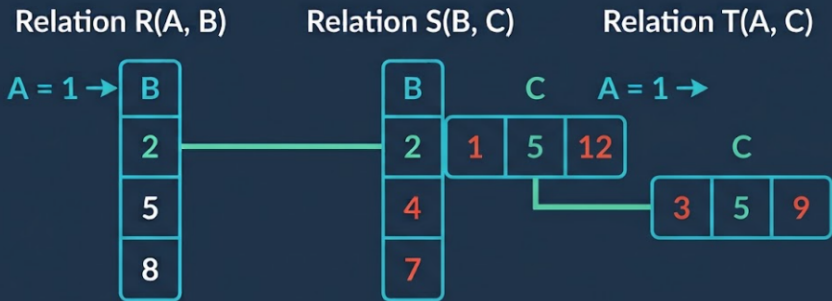


Worst case optimal joins

- A new join algorithm with guarantees on how bad it can get.
- Multi-way merge join that can skip non-matches
- Similar to how GIN handles popular-term - rare-term searches.



Query: $R(A, B) \bowtie S(B, C) \bowtie T(A, C)$



— = Match / Seek



That's it



Questions?

