Running Loops Around Recursive SQL

Daan de Graaf
Nikolay Yakovets

Altan Birler
Thomas Neumann

TU/e EINDHOVEN UNIVERSITY OF TECHNOLOGY

Technische Universität München

Recursion in SQL

- Loops are fundamental for many workloads
  - **Graph Analytics**
  - Data Mining
  - Machine Learning
- WITH RECURSIVE (since SQL:1999)
  - Theory: Turing Complete!
  - Practice: Hard to use and slow

```
WITH RECURSIVE reach AS (
   -- Base case
   SELECT 1 AS id
   UNION
   -- Recursive part
   SELECT edge.trg
   FROM edge, reach
   WHERE edge.src = reach.id
)
SELECT * FROM reach;
```

Reachability in SQL using WITH RECURSIVE

Problem I: Non-monotone Queries

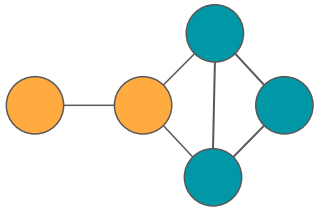
Monotone: Add tuples, but no update/delete

Consider Label Propagation Algorithm (CDLP):

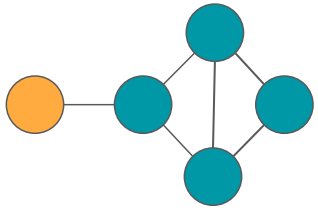
```
For each vertex:

1.  Count frequency of labels among neighbours
2.  Adopt most frequently occurring label
3.  Repeat until fixpoint
```

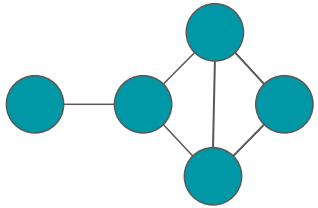
Iter 0



Iter 1



Iter 2



Problem II: Non-Linear Queries

Linear: A single reference to recursive table.

What does the following query output?

```
WITH RECURSIVE Ancestor AS (
       SELECT anc, des FROM Parent
       UNION
       SELECT a1.anc, a2.des
       FROM Ancestor a1, Ancestor a2
       WHERE a1.des = a2.anc)
SELECT * FROM Ancestor
```

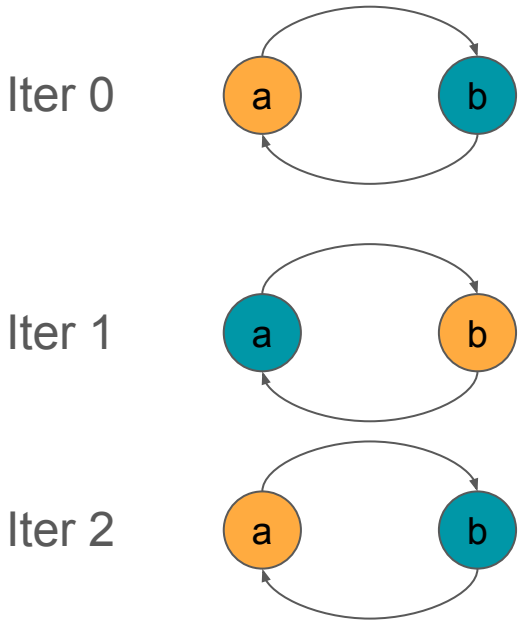
Parent	
ancestor	descendant
Alice	Bob
Bob	Charlie
Charlie	David

Problem III: Termination

Recall Label Propagation Algorithm (CDLP):

For each vertex:

1. Count frequency of labels among neighbours
2. Adopt most frequently occurring label
3. Repeat until fixpoint or iteration limit



Problem IV: Only one state table

- Multiple loop variables: Natural in most languages, **but not SQL**
- Workarounds hurt readability and performance

```
dist = {start: 0}        # Loop var 1
queue = deque([start])   # Loop var 2
while queue:
    cur = queue.popleft()
    for v in graph[cur]:
        if v not in dist:
            dist[v] = dist[cur] + 1
            queue.append(v)
```

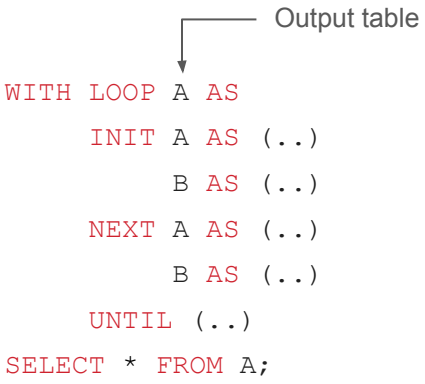
Breadth-First Search (BFS) in Python.

Problems and Workarounds

- Workarounds are possible, but they affect:
  - Readability
  - Execution time
  - Memory Consumption
- WITH RECURSIVE is bad for User and DBMS
  - Writing queries is hard
  - Optimizing inefficiencies away is hard
- Need a better abstraction

Introducing: **`WITH LOOP`**

- Replace or remove tuples (**Non-monotone**)
- Reference any state table (**Mutual and Non-linear recursion**)
- User-defined **break condition**



Syntax for WITH LOOP

Advanced Features

- In-place aggregation
- Convergence tracking
- Delta table

```
WITH LOOP dist
     INIT dist AS (SELECT 0 AS id, 0.0 AS dist)
     NEXT dist AS (SELECT trg AS id, dist + cost AS dist
                   FROM DELTA(dist) dist
                   JOIN edges ON (id = src))
     OPTIONS dist(KEY id,
                  AGGREGATE min(dist) AS dist,
                  CONVERGE)
```

Bellman-Ford (Single-Source Shortest Path) Algorithm.

WITH LOOP generalizes WITH RECURSIVE

Proposed flavours of SQL Recursion that are subsumed by WITH LOOP:

Listing 3: Standard Recursive CTE

```
WITH RECURSIVE T AS (
  q1 UNION ALL q∞(T))
```

Listing 4: Simulated Recursive CTE

```
WITH LOOP T
INIT T AS q1,
     T′ AS q∞(q1)
NEXT T AS T UNION ALL T′,
     T′ AS q∞(T′)
UNTIL SELECT COUNT(*) = 0
      FROM T′
```

Listing 5: Iterating CTE [18]

```
SELECT *
FROM ITERATE(q1, q∞, qu)
```

Listing 6: Simulated Iterating CTE

```
WITH LOOP iterate
     INIT iterate AS q1
     NEXT iterate AS q∞
     UNTIL qu
SELECT * FROM iterate
```

Listing 7: USING KEY [3]

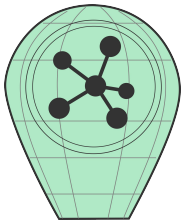
```
WITH RECURSIVE T(k1,...,km,c1,...,cn)
USING KEY (k1,...,km) AS (
  q1 UNION q∞(T, RECURRING T))
```

Listing 8: Simulated USING KEY

```
WITH LOOP T
INIT T AS q1
NEXT T AS q∞(DELTA T, T)
OPTIONS T (KEY (k1,...,km),
           CONVERGE)
```

Compiling GraphAlg[1] to SQL

- A language built for graph algorithms
- Designed to integrate into Database Systems
- Compiles to standard Relational Algebra, **except for loops!**



[1] de Graaf, et al. 2026. GraphAlg Playground: An Online Platform for Learning and Experimenting with the GraphAlg Language. Proc. VLDB Endow. 19, 12

Cypher

GraphAlg

```
WITH ALGORITHM "
func TriangleCount(graph: Matrix<s, s, bool>) -> int {
    L = tril(graph);
    U = triu(graph);
    C = Matrix<int>(graph.nrows, graph.ncols);
    C<L> = cast<int>(L) * cast<int>(U.T);
    return reduce(C);
}"
CALL TriangleCount()
RETURN count
```

Compiling GraphAlg to SQL

- Previous solution for loops: external driver script
  - Network Overhead
  - Limits Query Optimization
- `WITH LOOP` is the perfect compilation target

```
func BFS(graph: Matrix<s, s, bool>,
         source: Vector<s, bool>)
         -> Vector<s, int> {
  v = Vector<int>(graph.nrows);
  v<source>[:] = int(1);

  frontier = source;
  reach = source;
  for i in graph.nrows {
    step = Vector<bool>(graph.nrows);
    step<!reach> = frontier * graph;
    v<step>[:] = i + int(1);
    frontier = step;
    reach += step;
  } until frontier.nvals == int(0);

  return v;
}
```

Multiple Loop variables

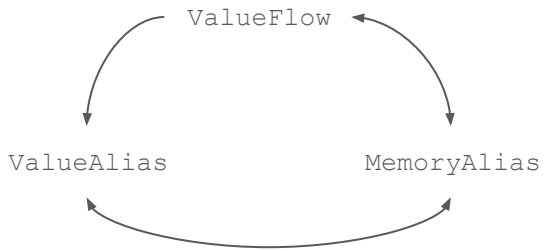
In-place aggregation

Break condition

Breadth-First Search (BFS) in GraphAlg.

Compiling Datalog to SQL

- Datalog engines are usually distinct from relational engines, why?
  - Set semantics
  - Nonlinear Recursion
  - Mutual Recursion
- `WITH LOOP` enables datalog support
  - Semi-naive execution: DELTA
  - Set semantics: KEY
  - Fixpoint: CONVERGE
  - Recursive aggregation: AGGREGATE



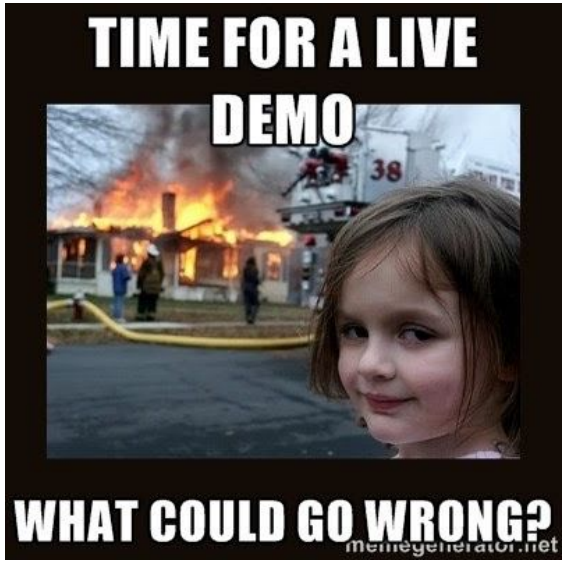

```
ValueFlow(x, y)    :- Assign(x, z),
                      MemoryAlias(z, y).
ValueFlow(x, y)    :- ValueFlow(x, z),
                      ValueFlow(z, y).
MemoryAlias(x, w)  :- Dereference(y, x),
                      ValueAlias(y, z),
                      Dereference(z, w).
ValueAlias(x, y)   :- ValueFlow(z, x),
                      ValueFlow(z, y).
ValueAlias(x, y)   :- ValueFlow(z, x),
                      MemoryAlias(z, w),
                      ValueFlow(w, y).
```

Context-Sensitive Points-to analysis in Datalog (subset of ruleset).

Compiling GraphAlg to SQL - Demo

https://wildarch.dev/graphalg/playground

https://umbra-db.com/interface/



Experimental Results

Workload	Soufflé	FlowLog	Umbra
CSPA (httpd)	49.7	24.9	18.8
Galen	31.6	10.0	4.7
DDISASM (cvc5)	19.1	17.1	16.3
DOOP (batik)	32.0	16.4	52.5

Datalog query execution time compared against Souffé and FlowLog.

Algorithm	System	wiki-Talk	cit-Patents	kgs	dota-league	graph500-22	datagen-7_8-zf	datagen-7_9-fb
BFS	Umbra (ext)	4.6	2.8	2.7	0.6	5.1	33.1	3.6
	Umbra (LOOP)	0.3	0.2	0.4	0.4	1.0	5.5	1.3
	DuckDB	0.5	0.7	1.0	0.8	2.0	6.8	2.7
	AvantGraph	1.1	0.8	1.0	0.4	1.6	29.3	1.0
CDLP	Umbra (ext)	2.4	4.7	3.0	4.2	7.3	14.0	14.5
	Umbra (LOOP)	1.0	3.4	2.0	3.2	6.5	8.4	13.1
	DuckDB	3.8	10.4	5.9	8.3	15.1	27.0	34.2
	AvantGraph	29.7	93.8	27.4	30.7	74.3	289.5	214.5
PR	Umbra (ext)	3.0	4.3	4.7	7.9	12.9	13.7	16.3
	Umbra (LOOP)	0.9	2.1	3.7	6.1	10.6	8.7	15.8
	DuckDB	2.2	4.9	5.2	10.0	15.5	17.6	20.5
	AvantGraph	2.4	3.9	2.4	4.2	7.5	16.3	8.3
SSSP	Umbra (ext)	*N/A*	*N/A*	7.1	7.9	*N/A*	55.1	5.1
	Umbra (KEY)			1.6	2.6		3.6	1.8
	Umbra (LOOP)			1.7	2.6		3.6	1.8
	DuckDB			5.4	5.6		36.4	4.9
	AvantGraph			6.0	14.2		34.5	4.4
WCC	Umbra (ext)	2.2	10.4	4.6	3.0	8.8	46.2	9.8
	Umbra (KEY)	0.4	1.8	0.8	1.0	2.8	4.1	4.4
	Umbra (LOOP)	0.6	3.4	1.3	0.8	2.9	12.0	2.9
	DuckDB	1.8	10.8	4.5	3.3	9.3	43.8	10.4
	AvantGraph	2.7	8.8	3.5	3.7	8.8	38.3	9.3

Processing times on LDBC Graphalytics (Up to scale S).

Future Work

- Adaptive Query Optimization
- Detect and apply advanced features using Query Optimizer
- Cheap joins over state tables
- Alternative storage strategies for loop state

Thank you!

Questions to the audience

- What can a graph DB do that a relational DB cannot?
- Should the goal be specialization or unification?

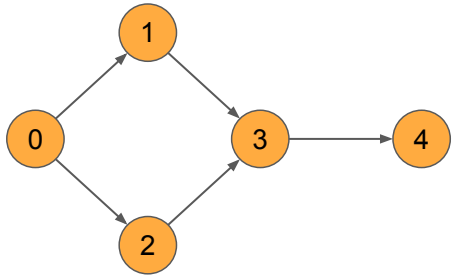
Bonus Slides

Integration into Umbra

- New algebra: `Loop`, `LoopStateScan`
- Storage for loop state
  - If `KEY` option: Hash table
  - Otherwise: Temp
- Building Delta table and Convergence check during Hash table insert
- Reused much of the infrastructure for WITH RECURSIVE, ITERATE and USING KEY
  - Iteration scopes (for pipeline restarting)
  - Hash table storage from USING KEY
  - Break condition from ITERATE
- Total diff: ~2000LOC

Example Algorithm: Reachability

```
WITH LOOP reach
    INIT reach AS (SELECT 0 AS id),
         front AS (SELECT 0 AS id)
    NEXT reach AS (SELECT id FROM reach
                   UNION
                   SELECT id FROM front),
         front AS (SELECT trg
                   FROM front
                   JOIN edges ON (id = src)
                   -- Anti-join with reach
                   WHERE NOT EXISTS (
                       SELECT 1
                       FROM reach
                       WHERE reach.id = trg))
  UNTIL (SELECT COUNT(*) = 0 FROM front)
```



reach	front
{0}	{0}
{0}	{1,2}
{0,1,2}	{3}
{0,1,2,3}	{4}
{0,1,2,3,4}	∅