A Fix for the Fixation on Fixpoints

CIDR
January 2023

Denis Hirn • Torsten Grust
University of Tübingen

23+ Years of Recursive CTEs in SQL

```
WITH RECURSIVE
T(…) AS (
  q1               -- initialize
    UNION ALL
  q∞(T)            -- iterate
)
TABLE T;
```

: *“Oh! What does this compute?”*

: *“The least* ***fixpoint*** $T = q_1$ **UNION ALL** $q_\infty(T)$*.”*

: …

23+ Years of Recursive CTEs in SQL

: *"The **fixpoint semantics** of CTEs serve SQL well."*

1. If query *q∞* is **monotonic**,
   the fixpoint does exist and is unique.

2. *q∞*'s monotonicity enables **semi-naive evaluation.**

: *"Uhm, that's good... right?"*

Monotonicity Leads to Syntactic Restrictions

```
WITH RECURSIVE
T(···) AS (
  q1

    UNION ALL

    ✕ NOT EXISTS      ✕ ORDER BY/LIMIT
    ✕ INTERSECT       ✕ DISTINCT
    ✕ EXCEPT          ✕ grouping
    ✕ outer joins     ✕ aggregation
)
TABLE T;
```

Monotonicity Zone

- Workarounds are part of the SQL developer folklore, yet often lead to nothing but syntactic atrocities. 😫

Semi-Naive Evaluation Leads to Short-Term Memory

```
WITH RECURSIVE
T(···) AS (
  q₁
    UNION ALL
  q∞(T)
     ↑      rows of immediately
     └───── preceding iteration
)
TABLE T;
```

TABLE T

pay	load	HISTORY
		[•]
		[•,•]
		[•,•,•]
		⋮
		[•,•,•,•,•,•,…,•]

- Query q_∞ cannot see the history of the computation (*e.g.*, visited nodes)
- Workaround: let q_∞ itself build/inspect **HISTORY** (potentially sizable)

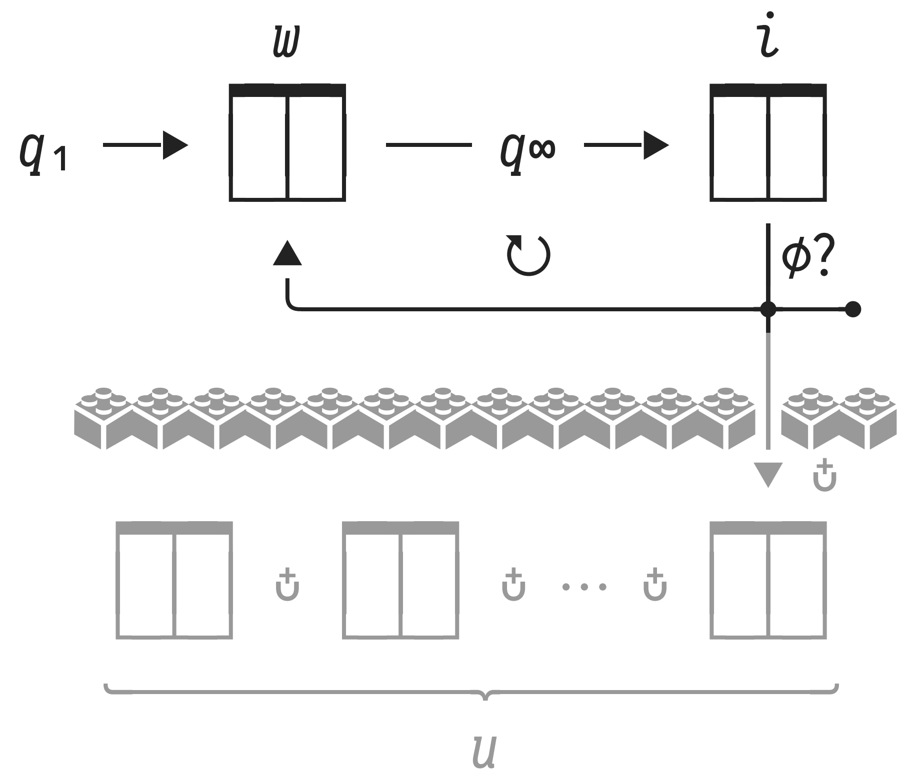
: *“Thank you, SQL folks.*
I'll keep using Python then.”

The Operational Loop-Based Semantics of WITH RECURSIVE

```
WITH RECURSIVE
T(c₁,…,cₙ) AS (
  q₁
    UNION ALL
  q∞(T)
)
TABLE T ;
```

- Found in most textbooks.
- Useful computational pattern, also if q_∞ is non-monotonic.
- Close to the actual engine-internal implementation.

```
u ← q₁
w ← u

loop
│ i ← q∞(w)
│ break if i = ∅
│ u ← u ⊎ i
│ w ← i
└
return u
```



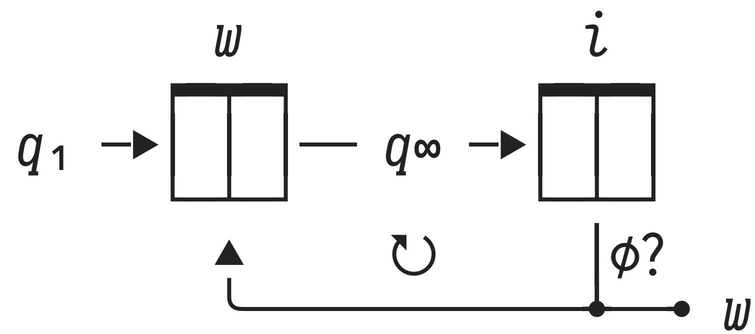
Tweaking the Operational Semantics: WITH ITERATIVE

```
WITH RECURSIVE
T(c₁,…,cₙ) AS (
  q₁
    UNION ALL
  q∞(T)
)
TABLE T ;
```

```
WITH ITERATIVE
T(c₁,…,cₙ) AS (
  q₁
    UNION ALL
  q∞(T)
)
TABLE T ;
```

```
u ← q₁
w ← u

loop
│ i ← q∞(w)
│ break if i = ϕ
│ u ← u ⊎ i
│ w ← i
└
return u
```

```
w ← q₁

loop
│ i ← q∞(w)
│ break if i = ϕ
│ w ← i
└
return w -- q∞'s last non-empty result
```



A Fix for the Fixation on Fixpoints: New CTE Variants

- Start from the operational semantics for **WITH RECURSIVE:**
  - Aim for **simple, loop-based** CTE behavior.
  - Leverage **existing CTE infrastructure.**

- **Lift** fixpoint-induced monotonicity **restrictions** on q_∞.

❶ WITH ITERATIVE … KEY

- operate table *u* like an updatable keyed dictionary
- keys control size of dict
- q_∞ can read entire dict

❷ WITH ITERATIVE … TTL

- q_∞ sees results of ⩾ 1 earlier iterations
- results age, then expire
- non-linear recursion OK

CTE Variant ❶: Operate Table *u* Like a Keyed Dictionary

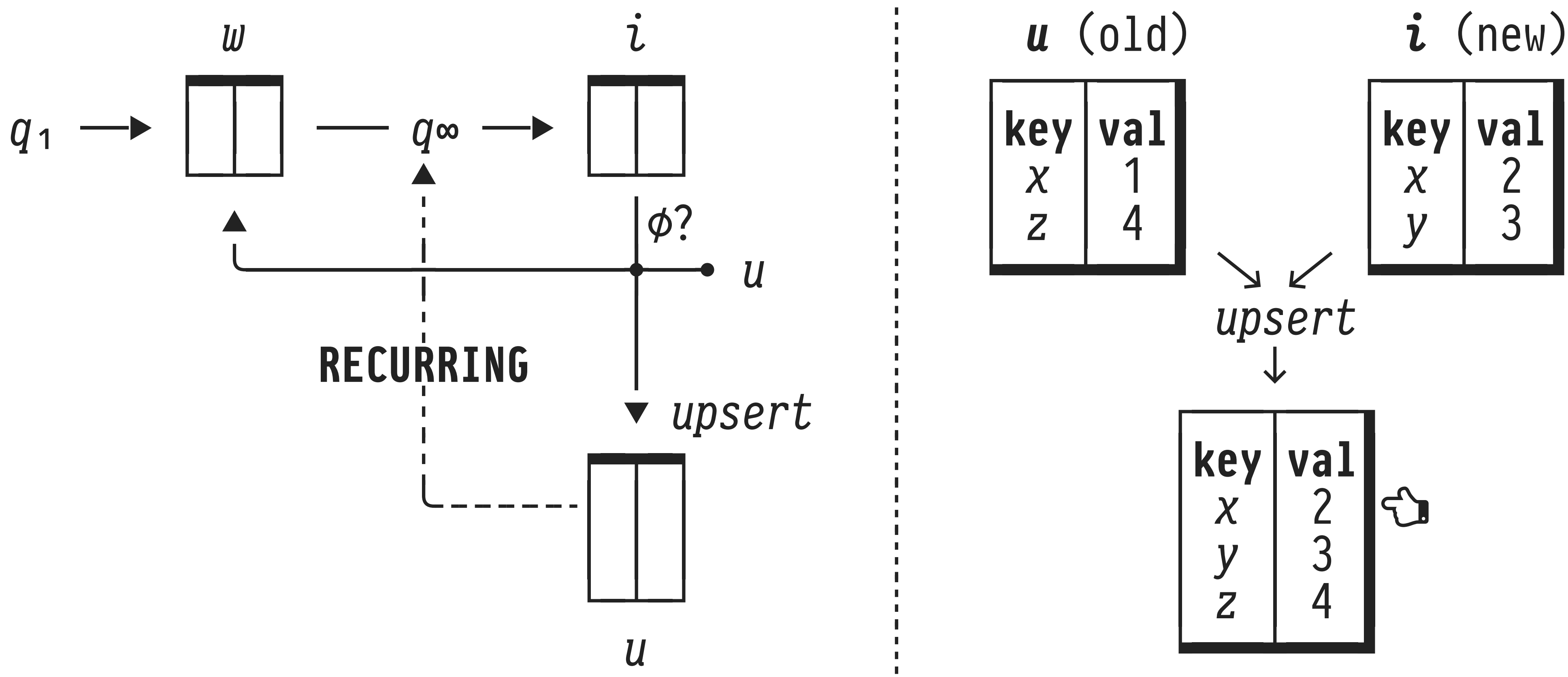
```
WITH RECURSIVE
T(c₁,…,cₙ) AS (
  q₁
    UNION ALL
  q∞(T)
)
TABLE T ;
```

```
WITH ITERATIVE
T(k₁,…,kₘ,c₁,…,cₙ) KEY (k₁,…,kₘ) AS (
  q₁
    UNION ALL
  q∞(T, RECURRING(T) )
)
TABLE T ;
```

```
u ← q₁
w ← u

loop
│ i ← q∞(w)
│ break if i = ϕ
│ u ← u ∪̇ i
│ w ← i
└
return u
```

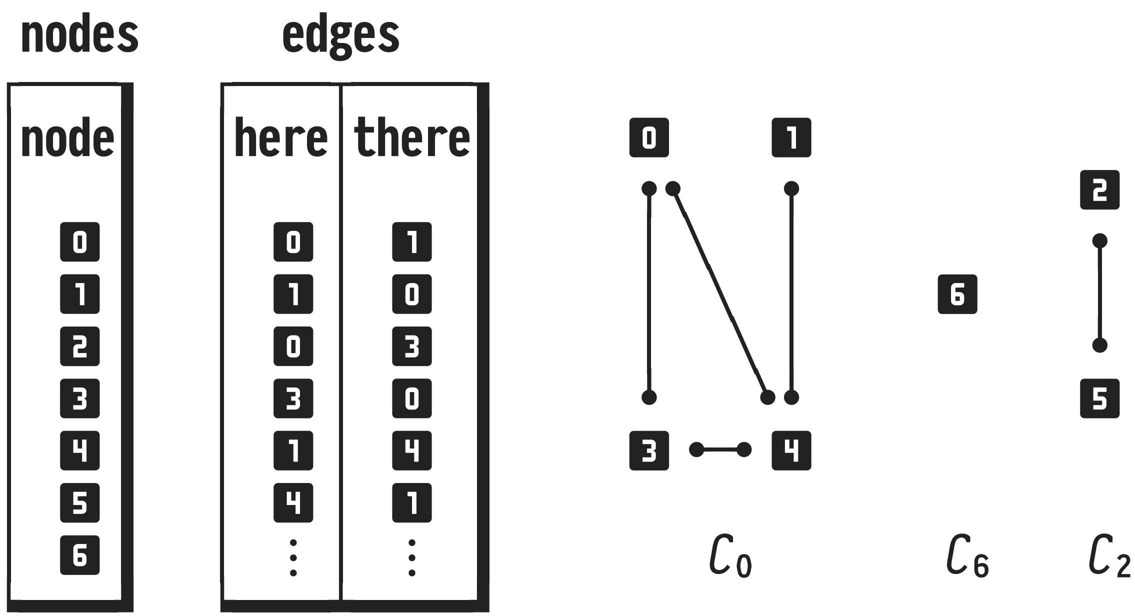
```
u ← upsert(ϕ, q₁)
w ← u

loop
│ i ← q∞(w, u )
│ break if i = ϕ
│ u ← upsert(u, i)
│ w ← i
└
return u
```

CTE Variant ❶: Operate Table *u* Like a Keyed Dictionary



- Operate union table *u* like a **keyed dictionary.**
- q_∞ has access to "hot rows" *and* dictionary **RECURRING**(*T*).
- Active domain of column **key** controls dictionary size.
- 💡 Refer to/update the dictionary like **an imperative PL.**

Exercising CTE Variant ❶: Connected Graph Components



Components C_i

- Find the **connected components** of an undirected graph: build array cc[0] = C_0, cc[1] = C_0, cc[2] = C_2, …

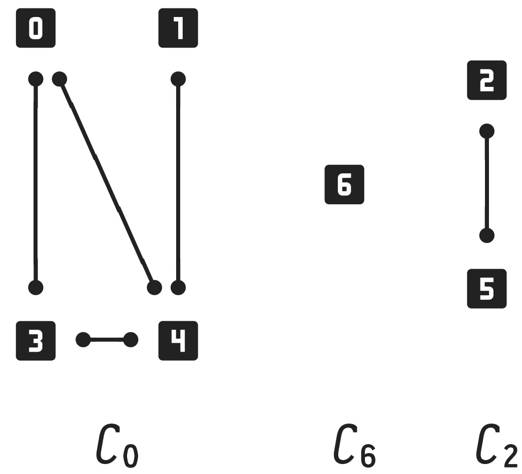
Exercising CTE Variant ❶: Connected Graph Components

nodes

node
0
1
2
3
4
5
6

edges

here	there
0	1
1	0
0	3
3	0
1	4
4	1
⋮	⋮



Components C_i

cc

<u>node</u>	comp
0	0
1	1
2	2
3	3
4	4
5	5
6	6

❶ **initialize:**
$\forall\ n \in nodes$: cc[n] ← n;

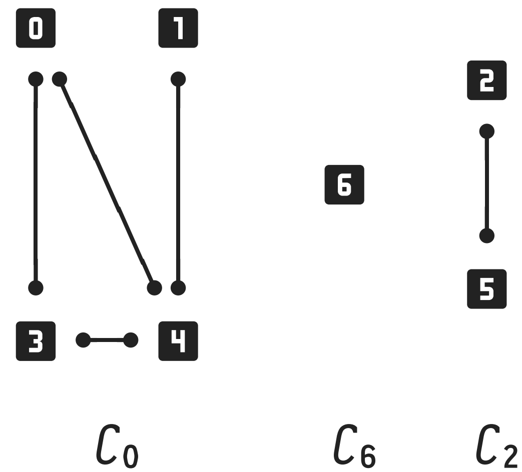
Exercising CTE Variant ❶: Connected Graph Components

nodes

node
0
1
2
3
4
5
6

edges

here	there
0	1
1	0
0	3
3	0
1	4
4	1
⋮	⋮



Components C_i

cc

node	comp
0	0
1	1
2	2
3	0
4	0
5	2
6	6

❷ **iterate** and **update:**

cc[u] ← min { cc[v] | u •——• v };

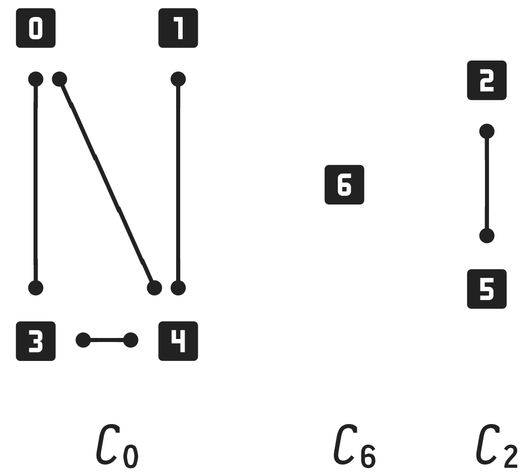
Exercising CTE Variant ❶: Connected Graph Components

nodes

node
0
1
2
3
4
5
6

edges

here	there
0	1
1	0
0	3
3	0
1	4
4	1
⋮	⋮



Components C_i

cc

<u>node</u>	comp
0	0
1	0
2	2
3	0
4	0
5	2
6	6

❷ **iterate** and **update:**
cc[u] ← min { cc[v] | u •——• v };

Exercising CTE Variant ❶: Connected Graph Components

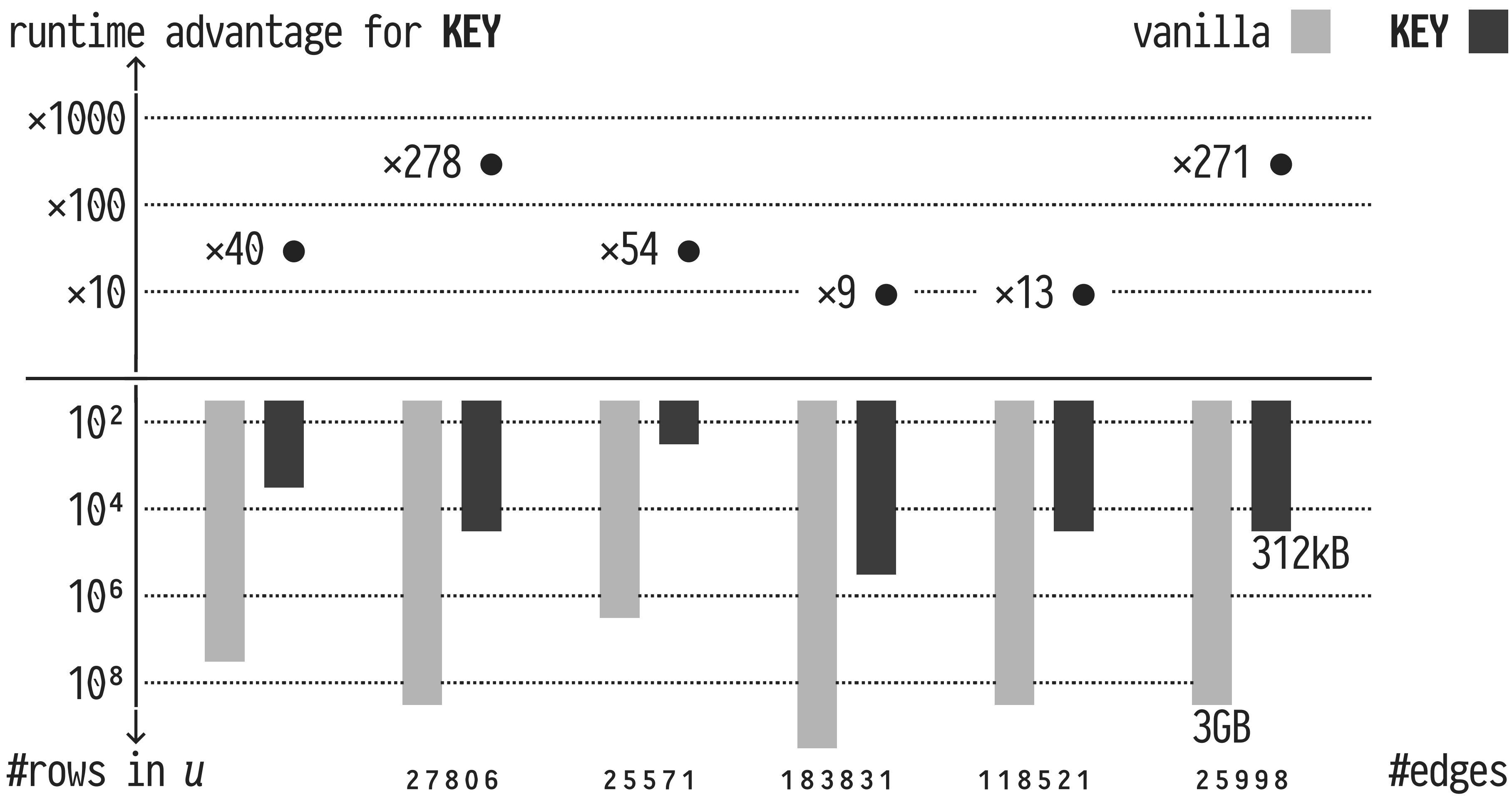
Aim to transcribe the folklore stateful algorithm directly into SQL:

```
foreach n in nodes
⌊ cc[n] ← n

while true

│ N ← updated nodes
│ if N = ∅ then return cc
│
│ foreach key u in cc, v in N
│ │ foreach u•—•v in edges
│ │ │ if cc[v] < cc[u] then
⌊ ⌊ ⌊ ⌊ cc[u] ← cc[v]
```

```
WITH ITERATIVE
cc(node, comp) KEY (node) AS (
  SELECT n.node, n.node AS comp
  FROM   nodes AS n

    UNION ALL

 (SELECT DISTINCT ON (node) u.node, v.comp
  FROM   RECURRING(cc) AS u, cc AS v, edges AS e
  WHERE  (e.here,e.there) = (u.node,v.node)
  AND    v.comp < u.comp
  ORDER BY u.node, v.comp)
)
TABLE cc;
```

☞ q_∞ emits ‹node,comp› ≡ array update cc[node] ← comp.

WITH ITERATIVE … KEY vs. Vanilla WITH RECURSIVE



- **WITH ITERATIVE…KEY:** table *u* always holds ⩽ |**nodes**| rows.

CTE Variant ❷: Aging Row Memory

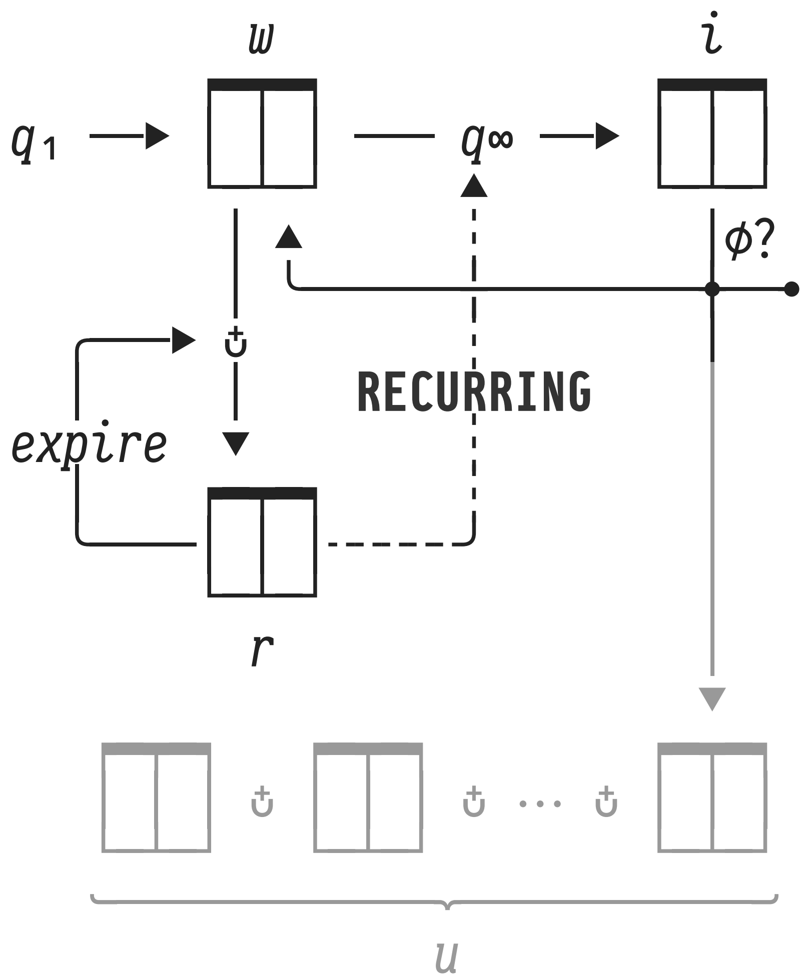
```
WITH RECURSIVE
T(c₁, … ,cₙ) AS (
  q₁
    UNION ALL
  q∞(T)
)
TABLE T ;
```

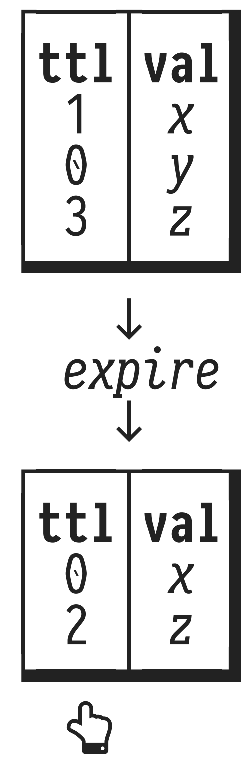
```
u ← q₁
w ← u

loop
│ i ← q∞(w)
│ break if i = ϕ
│ u ← u ⊎ i
│ w ← i
└
return u
```

```
WITH ITERATIVE
T(ttl,c₁, … ,cₙ) TTL (ttl) AS (
  q₁
    UNION ALL
  q∞(T, RECURRING(T) )
)
TABLE T ;
```

```
u ← q₁
w ← expire(u)
r ← w
loop
│ i ← q∞(w, r )
│ break if i = ϕ
│ u ← u ⊎ i
│ w ← expire(i)
└ r ← expire(r) ⊎ w
return u
```

CTE Variant ❷: Aging Row Memory





- Former results accessible during their "*time to live*."
- **ttl** set as needed by q_∞—controls size of **RECURRING**(T).

Exercising CTE Variant ❷: CYK Parsing

Given context-free grammar in Chomsky normal form (CNF), parse string (👍/👎 or build parse tree):

$$
\begin{aligned}
\mathit{Exp} &\to \mathit{Sum}\ \mathit{Term} \mid \mathit{Prod}\ \mathit{Fact} \mid [0..9]^{+} \mid \mathit{Sub}\ \mathit{Close} \\
\mathit{Term} &\to \mathit{Prod}\ \mathit{Fact} \mid [0..9]^{+} \mid \mathit{Sum}\ \mathit{Term} \\
\mathit{Sub} &\to \mathit{Open}\ \mathit{Exp} \\
\mathit{Sum} &\to \mathit{Exp}\ \mathit{Add} \\
\mathit{Prod} &\to \mathit{Term}\ \mathit{Mul} \\
\mathit{Fact} &\to [0..9]^{+} \mid \mathit{Sub}\ \mathit{Close} \\
\mathit{Open} &\to (\\
\mathit{Close} &\to) \\
\mathrm{Mul} &\to \times \\
\mathrm{Add} &\to +
\end{aligned}
$$

Input: 6 × (3 + 4) **OK** ✓

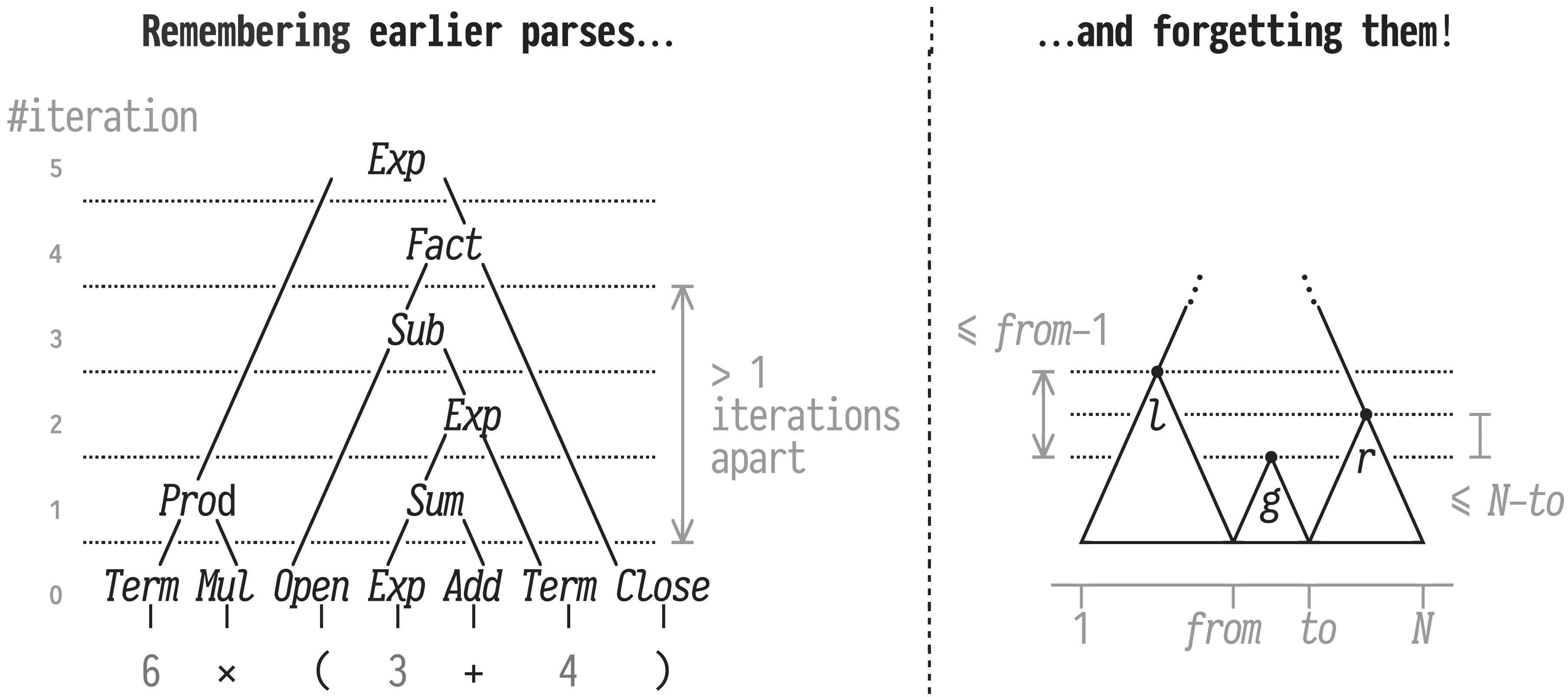
grammar

lhs	sym	rhs1	rhs2
Exp	□	*Sum*	*Term*
Exp	□	*Prod*	*Fact*
Exp	$[0..9]^{+}$	□	□
⋮	⋮	⋮	⋮
Add	[+]	□	□

- Regular CNF facilitates tabular grammar representation.

Exercising CTE Variant ❷: CYK Parsing

- Iterations build parse tree bottom-up.
- Remembering one preceding iteration only is *not enough*:



- 💡 Can limit **ttl** for parse *g*:
  will join with parses *l* or *r* once these have been built.

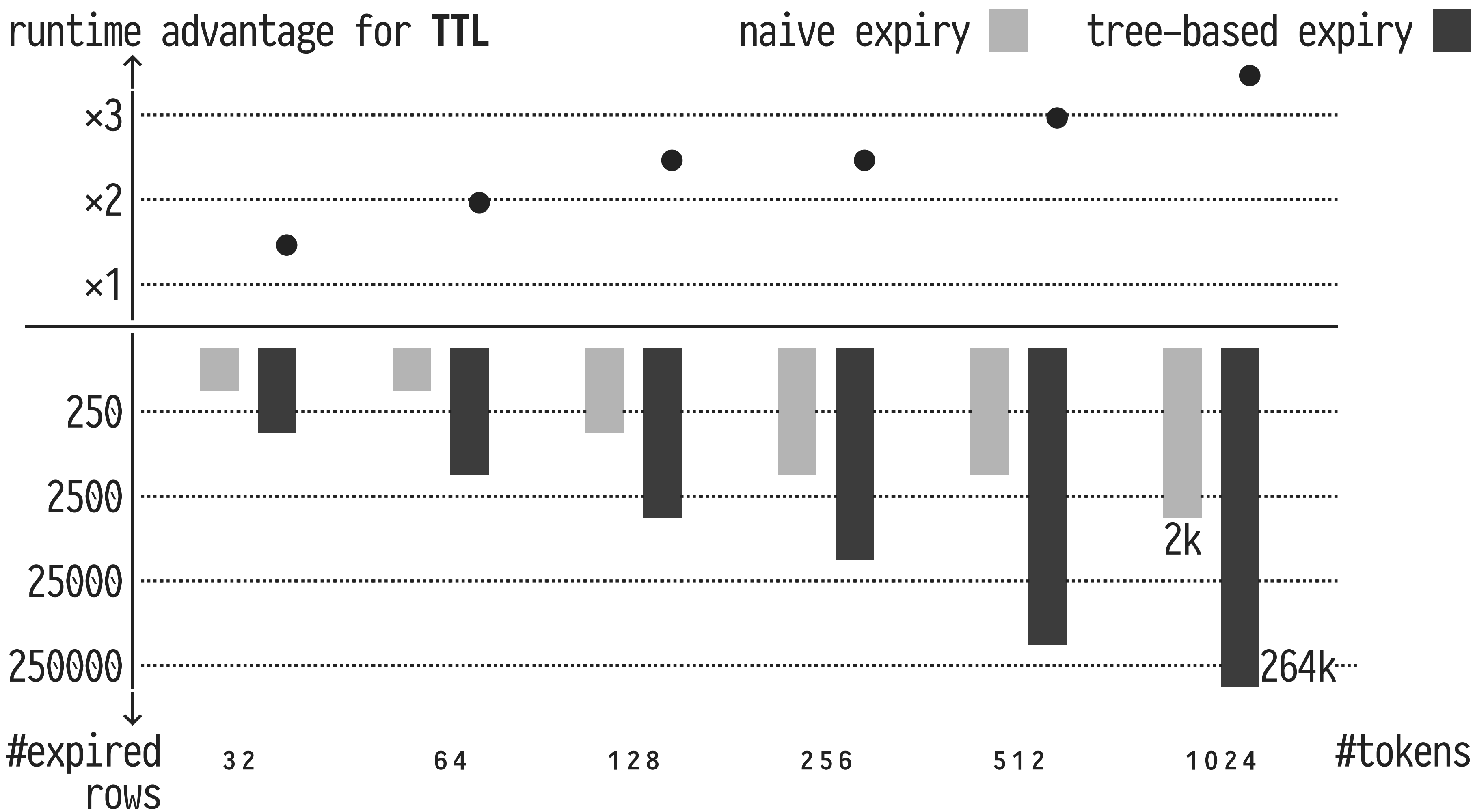
Exercising CTE Variant ❷: CYK Parsing

An 8-liner SQL query implements a CYK parser:

```
WITH ITERATIVE
parse(ttl, lhs, from, to) TTL (ttl) AS (
  SELECT GREATEST(t.i-1, N-t.i) AS ttl , g.lhs, t.i AS from, t.i AS to
  FROM   tokens AS t, grammar AS g
  WHERE  t.sym ~ g.sym

   UNION     keep parses only as needed

  SELECT GREATEST(l.from-1, N-r.to) AS ttl , g.lhs, l.from, r.to
  FROM   RECURRING(parse) AS l, RECURRING(parse) AS r, grammar AS g
  WHERE  l.to+1 = r.from
  AND    (g.rhs1, g.rhs2) = (l.lhs, r.lhs)
)
```

- Tree-individual **ttl** keeps size of **RECURRING**(parse) down.
- Selective row memory makes **non-linear recursion** viable.

Controlled Row Expiry Helps Non-Linear Recursion



- **TTL**-based expiry vs. manual row "reinjection" (in 🐘).

More Fixes for the Fixation on Fixpoints

- Reach into RDBMS CTE code to optimize run time ⏱:
  - **KEY:** large dicts based on hashing infrastructure.
  - **TTL:** speed up row expiry via an **ttl**-based queue.
- Beyond variants **KEY** and **TTL:**

1. Let q^∞ place rows in one of **multiple working tables.**
2. More modifiers like **RECURRING(·)** that return rows using a LIFO discipline (**working stack**).
3. CTE variants that can serve as **compilation targets** for iterative PL/SQL code.

A Fix for the Fixation on Fixpoints

CIDR

January 2023

Denis Hirn • Torsten Grust

University of Tübingen

@Teggy | db.cs.uni-tuebingen.de/grust