A Fix for the Fixation on Fixpoints
CIDR
January 2023
Denis Hirn • Torsten Grust
University of Tübingen
A Fix for the Fixation on Fixpoints

CIDR
January 2023

Denis Hirn • Torsten Grust
University of Tübingen
WITH RECURSIVE
T(...) AS ( 
    q₁       -- initialize
    UNION ALL
    q∞(T)    -- iterate
  )
TABLE T;

Emoji: "Oh! What does this compute?"

Hat: "The least fixpoint $T = q₁ \text{ UNION ALL } q∞(T)$.”

Smiley: ...
읶: “The fixpoint semantics of CTEs serve SQL well.”

1️⃣ If query $q^\infty$ is monotonic, the fixpoint does exist and is unique.

2️⃣ $q^\infty$'s monotonicity enables semi-naive evaluation.

[Math]: “Uhm, that's good... right?”
WITH RECURSIVE
T(⋯) AS ( 

$q_1$

UNION ALL

× NOT EXISTS × ORDER BY/LIMIT
× INTERSECT × DISTINCT
× EXCEPT × grouping
× outer joins × aggregation

) TABLE T;

• Workarounds are part of the SQL developer folklore, yet often lead to nothing but syntactic atrocities. 😞
WITH RECURSIVE
T(...) AS ( 
    q₁
    UNION ALL
    q∞(T)
    rows of immediately preceding iteration
) TABLE T;

TABLE T

<table>
<thead>
<tr>
<th>pay</th>
<th>load</th>
<th>HISTORY</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>[•]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>[•,•]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>[•,•,•]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>[•,...]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>[•,•,...]</td>
</tr>
</tbody>
</table>
∞️: “Thank you, SQL folks. I'll keep using Python 🐍 then.”
The Operational Loop-Based Semantics of WITH RECURSIVE

WITH RECURSIVE

\[
T(c_1, \ldots, c_n) \text{ AS (}
\begin{align*}
q_1 \\
\quad \text{UNION ALL} \\
q_\infty(T)
\end{align*}
\)
\]

TABLE \(T\);

\[
1 \quad u \leftarrow q_1
\]
\[
2 \quad w \leftarrow u
\]

\[
\begin{align*}
\text{loop} \\
4 \quad i \leftarrow q_\infty(w) \\
5 \quad \text{break if } i = \phi \\
6 \quad u \leftarrow u \uplus i \\
7 \quad w \leftarrow i
\end{align*}
\]

\[
10 \quad \text{return } u
\]

- Found in most textbooks.
- Useful computational pattern, also if \(q_\infty\) is non-monotonic.
- Close to the actual engine-internal implementation.
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;

1 u ← q₁
2 w ← u

loop
4 i ← q∞(w)
5 break if i = φ
6 u ← u ∪ i
7 w ← i
8 return u

10 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 $q^\infty$.

1. **WITH ITERATIVE ... KEY**
   ◦ operate table $u$ like an updatable keyed dictionary
   ◦ keys control size of dict
   ◦ $q^\infty$ can read entire dict

2. **WITH ITERATIVE ... TTL**
   ◦ $q^\infty$ sees results of $\geq 1$ earlier iterations
   ◦ results age, then expire
   ◦ non-linear recursion OK
## CTE Variant 1: Operate Table $u$ Like a Keyed Dictionary

### WITH RECURSIVE

```sql
WITH RECURSIVE
T($c_1,...,c_n$) AS (  
q_1
  UNION ALL
  $q_\infty(T)$
)
TABLE T;
```

1. $u \leftarrow q_1$
2. $w \leftarrow u$

### loop

```sql
loop
  i \leftarrow q_\infty(w)
  break if i = \phi
  u \leftarrow u \cup i
  w \leftarrow i
return u
```

### WITH ITERATIVE

```sql
WITH ITERATIVE
T($k_1,...,k_m,c_1,...,c_n$) KEY ($k_1,...,k_m$) AS (  
q_1
  UNION ALL
  $q_\infty(T, RECURRING(T))$
)
TABLE T;
```

1. $u \leftarrow \text{upsert}(\phi, q_1)$
2. $w \leftarrow u$

### loop

```sql
loop
  i \leftarrow q_\infty(w, u )
  break if i = \phi
  u \leftarrow \text{upsert}(u, i)
  w \leftarrow i
return u
```
CTE Variant ①: Operate Table \( u \) Like a Keyed Dictionary

- Operate union table \( u \) like a keyed dictionary.
- \( q^\infty \) has access to “hot rows” and dictionary \textsc{Recurring}(\( T \)).
- Active domain of column \texttt{key} controls dictionary size.
- \( \circ \) Refer to/update the dictionary like an imperative PL.
Exercising CTE Variant ➊: Connected Graph Components

- 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 ①: Connected Graph Components

<table>
<thead>
<tr>
<th>Nodes</th>
<th>Edges</th>
<th>CC</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Nodes:</strong></td>
<td><strong>Edges:</strong></td>
<td><strong>CC:</strong></td>
</tr>
<tr>
<td>node</td>
<td>here</td>
<td>there</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>6</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

- **Components $C_i$:**
  - $C_0$
  - $C_6$
  - $C_2$

#### ① Initialize:
\[ \forall n \in \text{nodes}: \text{cc}[n] \leftarrow n; \]
Exercising CTE Variant ①: Connected Graph Components

② iterate and update:

\[
cc[u] \leftarrow \min \{ cc[v] \mid u \rightarrow v \};
\]
Exercising CTE Variant 1: Connected Graph Components

iterate and update:
\[ cc[u] \leftarrow \min \{ cc[v] \mid u \to v \}; \]
Exercising CTE Variant 1: Connected Graph Components

Aim to transcribe the folklore stateful algorithm directly into SQL:

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

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] ]

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;
```

$q^\circ$ emits $\langle$node,comp$\rangle$ ≡ array update cc[node] ← comp.
• WITH ITERATIVE...KEY: table $u$ always holds $\leq |\text{nodes}|$ rows.
CTE Variant ②: Aging Row Memory

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

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

1  u ← q₁
2  w ← u

loop
4  i ← q∞(w)
5  break if i = φ
6  u ← u  i
7  w ← i
9  return u

1  u ← q₁
2  w ← expire(u)
3  r ← w

loop
4  i ← q∞(w, r)
5  break if i = φ
6  u ← u  i
7  w ← expire(i)
8  r ← expire(r)  w
10 return u
CTE Variant ②: Aging Row Memory

- Former results accessible during their “time to live.”
- \( \text{ttl} \) set as needed by \( q^\infty \)—controls size of \( \text{RECURRING}(T) \).
Exercising CTE Variant ②: CYK Parsing

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

\[
\begin{align*}
\text{Exp} &\rightarrow \text{Sum} \text{ Term} \\
\text{Sum} &\rightarrow \text{Exp} \text{ Add} \\
\text{Prod} &\rightarrow \text{Fact} \\
\text{Fact} &\rightarrow [0..9]^+ \\
\text{Sub} &\rightarrow \text{Close} \\
\text{Term} &\rightarrow \text{Prod} \text{ Fact} \\
\text{Open} &\rightarrow ( \\
\text{Close} &\rightarrow ) \\
\text{Mul} &\rightarrow \times \\
\text{Add} &\rightarrow +
\end{align*}
\]

<table>
<thead>
<tr>
<th>grammar</th>
</tr>
</thead>
<tbody>
<tr>
<td>lhs</td>
</tr>
<tr>
<td>-------</td>
</tr>
<tr>
<td>Exp</td>
</tr>
<tr>
<td>Exp</td>
</tr>
<tr>
<td>Exp</td>
</tr>
<tr>
<td>Add</td>
</tr>
</tbody>
</table>

Input: \( 6 \times (3 + 4) \) OK ✓

- Regular CNF facilitates tabular grammar representation.
Exercising CTE Variant 2: CYK Parsing

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

```
WITH ITERATIVE
    b
    (d,tl,lh,lo)
    AS (SELECT 
        t.i-1, t.i AS tl, g.lhs, t.i AS rom, t.i AS do
        FROM tokens AS t, grammar AS g
        WHERE t.sym ~ g.sym
        UNION
        SELECT 
        t.i-1, t.i AS tl, g.lhs, t.i, t.i
        FROM RECURRING
        (b)
        AS l,
        RECURRING
        (b)
        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 (parse) down.
Selective row memory makes non-linear recursion viable.

- Can limit ttl for parse g: will join with parses l or r once these have been built.
Exercising CTE Variant ②: CYK Parsing

An 8-liner SQL query implements a CYK parser:

```sql
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
```

```sql
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.
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 a **ttl**-based queue.

Beyond variants **KEY** and **TTL**:

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

---

**Controlled Row Expiry Helps Non-Linear Recursion**

- **TTL**-based expiry vs. manual row “reinjection” (in 🗺️).

![Graph showing runtime advantage for TTL](image)
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 \( q^\infty \) place rows in one of multiple working tables.
  2. More modifiers like `RECURRING(\cdot)` that return rows using a LIFO discipline (working stack).
  3. CTE variants that can serve as compilation targets for iterative PL/SQL code.