A Fix for the Fixation on Fixpoints

Denis Hirn Torsten Grust
University of Tübingen
Tübingen, Germany
[denis.hirn,torsten.grust]@uni-tuebingen.de

ABSTRACT

We derive new iterative CTE variants from the simple loop-based, operational semantics of SQL:1999's `WITH RECURSIVE`. In the absence of fixpoint-based semantics and monotonicity restrictions, these CTE variants (1) can hold onto as well as forget the results of earlier loop iterations or (2) maintain iteration results in a keyed table, enabling a SQL authoring style that mimics imperative algorithms. We exercise the new variants using a series of examples to demonstrate that this fresh look at CTEs has a beneficial impact on the readability and performance of iterative SQL queries.

1 ITERATE LIKE IT'S 1999?

SQL:1999 [7, 20] introduced `WITH RECURSIVE`—or *recursive common table expressions* (CTEs)—a true game changer which turned the query language into a Turing-complete programming language with, admittedly, a very distinctive flavor. In a nutshell, the recursive CTE on the left iterates the evaluation of query q_{∞} (read: "q *loop*") which can refer to table T to access the rows produced in an earlier iteration.

```
WITH RECURSIVE
T(···) AS (
  q1
    UNION [ALL]
  q∞(T)
)
TABLE T;
```

The first iteration of q_{∞} processes the rows produced by q_1. Iteration stops once the least *fixpoint* $T = q_1$ `UNION [ALL]` $q_{\infty}(T)$ has been reached, returning table T as the overall result.

`WITH RECURSIVE` is powerful and versatile but proved to be notoriously hard to grasp and master. Indeed, only simple applications like the computation of transitive closures or bills of material in hierachical assemblies prevail in practice. Among the many challenges, let us highlight the following:

(1) The existence and uniqueness of the fixpoint is guaranteed only if q_{∞} is monotonic [2, 8]. This leads to significant **syntactic restrictions** on q_{∞} that rule out uses of negation (*e.g.*, `NOT EXISTS`), `INTERSECT`/`EXCEPT`, outer joins, duplicate row elimination via `DISTINCT`, or grouping and aggregation. Working around such limitations can turn simple, idiomatic SQL queries into syntactic atrocities.

(2) Monotonicity enables the semi-naive evaluation of q_{∞} over only the rows produced in the *immediately preceding iteration* [1]. While this aids the efficient execution of recursive CTEs, q_{∞} now exhibits **"short-term memory"** in which prior results (or the history of the computation, if you will) are inaccessible. This led query authors to adopt idioms that collect result rows in array-like structures to be carried from iteration to iteration [17, § 7.8.2.2, `CYCLE`]. Not only does this clutter the query code, but such home-grown row memory management comes at a space and runtime price. On top of that, semi-naive evaluation requires q_{∞} to be **linear** in T [2], adding further to the pile of syntactic restrictions.

In consequence, we find complex iterative computation that could benefit from evaluation close to the tabular data to be (i) predominantly realized outside RDBMSs [4, 11, 13] or (ii) cast in terms of iterative or recursive PL/SQL or SQL UDFs. Both options come with their own performance drawbacks [5, 12].

Iterative CTEs based on simple loops. To counter this frustrating state of affairs, the following pages explore descendants of `WITH RECURSIVE` that directly derive from its straightforward *loop-based, operational semantics.*

- This loop-based semantics explains the behavior of iterative CTEs in a procedural style as it is typically found in RDBMS documentation or textbooks. Unlike fixpoints, this semantics should be immediately comprehensible for query authors (see Section 2).
- In the absence of fixpoint-induced monotonicity requirements, we may lift all syntactic restrictions on q_{∞}, leading to compact (even elegant) query code that requires fewer workarounds.
- We discuss a CTE variant in which table T is operated like a keyed dictionary of rows that admits to read and overwrite former result rows. Such dictionaries (or associative arrays) are core data structures in many imperative algorithms, admitting a direct transcription of these algorithms into SQL (Section 3.1).
- We promote selective long-term memory in which queries control whether or how long a result row shall be remembered and thus be available in future iterations. This also enables sensible uses of non-linear references to T in q_{∞} (Sections 3.2 and 3.3).
- The loop-based semantics purposely mimics the actual implementations of recursive CTEs in database kernels. The proposed CTE variants are thus easily integrated into existing query engines.

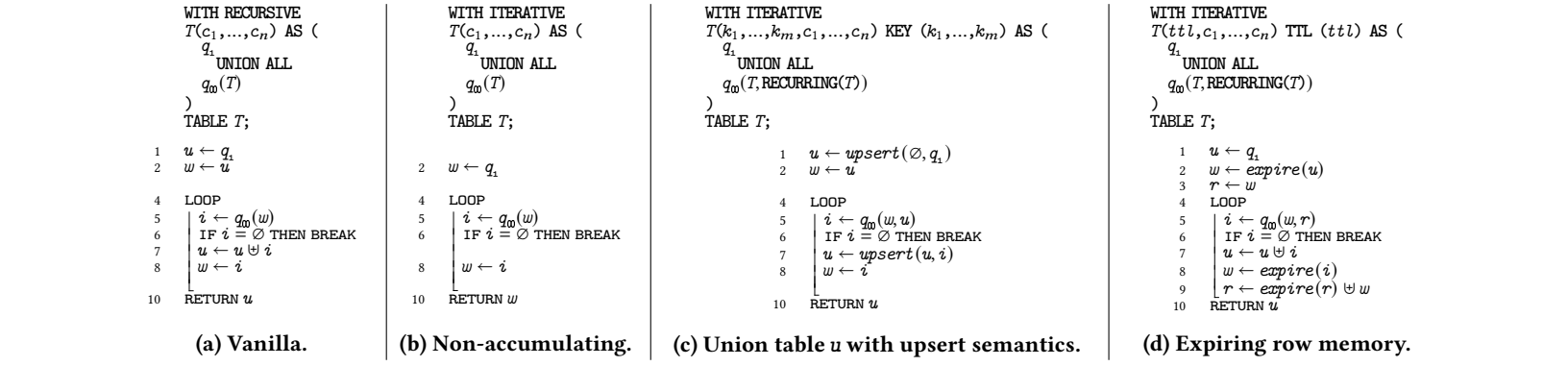
We dedicate the lion share of the paper to sketch scenarios that test-drive these new iterative SQL CTEs. Section 3 reviews the resulting query code but also sheds lights on its runtime and space usage.

Let us not proceed without noting that we are *not* keen to promote new SQL syntax. However, we invite readers to dabble with the thought of how iterative queries in SQL could evolve or be different from what was proposed 23 years ago.

2 CTEs THAT LOOP

A recursive CTE evaluates the initial SQL query q_1 once, then iterates the evaluation of q_{∞}. This essence is captured by the loop-based, procedural account of `WITH RECURSIVE` in Figure 1a. From this original loop, we derive a family of iterative CTE constructs that preserve this essence (Figures 1b to 1d). The roles of the table-valued variables w, i, u in all four CTE variants coincide, only table r is newly introduced to realize longer-term row memory:

 13th Annual Conference on Innovative Data Systems Research (CIDR '23). January 8-11, 2023, Amsterdam, The Netherlands.



(a) Vanilla. **(b) Non-accumulating.** **(c) Union table u with upsert semantics.** **(d) Expiring row memory.**

Figure 1: Loop-based, operational semantics for a family of iterative SQL CTEs. Our focus is on the KEY and TTL variants.

$$upsert(u,i) \equiv \begin{cases} \text{key error} & \text{if } |\delta(\pi_{k_1,\ldots,k_m}(i))| < |i| \\ (u \ltimes_{k_1,\ldots,k_m} i) \cup i & \text{otherwise} \end{cases}$$

(a) Maintaining union table u in CTE variant KEY ($\ltimes$ and δ denote left antijoin and duplicate elimination, respectively).

$$expire(s) \equiv \pi_{ttl \leftarrow ttl-1,c_1,\ldots,c_n}\left(\sigma_{ttl>0}(s)\right)$$

(b) Row expiry and aging in the TTL CTE variant.

Figure 2: Auxiliaries to support CTE variants KEY and TTL.

- w (*working table*): holds the rows produced by the immediately preceding iteration. Accessible by q_∞ through table name T.
- i (*intermediate table*): holds the rows of the current evaluation of q_∞. All CTE variants exit their LOOP if i turns out to be empty (see Lines 5 and 6 in Figure 1).
- u (*union table*): collects rows returned by q_1 and all intermediate tables computed by q_∞. Defines the CTE's result.
- r (*recurring table*): holds rows produced by earlier iterations (up to a defined row age), providing controlled access to the history of the computation. Accessible by q_∞ through RECURRING(T).

Let us shine a light on all CTE variants.

Vanilla WITH RECURSIVE. We argue that the procedure of Figure 1a embodies the intuitive understanding that most query authors have developed for recursive CTEs. If q_∞ is monotonic, the loop meets the original SQL:1999 fixpoint semantics.

Importantly, (i) the utility of this iterative computational pattern does *not* hinge on q_∞ being monotonic, and (ii) the loop closely matches the engine-internal implementations of WITH RECURSIVE (this is certainly so for PostgreSQL [16]).[1] The basic loop of Figure 1a thus makes for the ideal jumping-off point for our exploration.

Once we shake off the fixpoint prerequisite, we can obtain three interesting CTE variants through few local loop code changes that, nevertheless, still honor the original CTE spirit.

Non-accumulating WITH ITERATIVE (Figure 1b). Originally proposed in [15] to efficiently express in-database clustering over tabular data, WITH ITERATIVE does not accumulate intermediate results in union table u (the loop thus misses the assignments to u in Lines 1 and 7). Since the last non-empty evaluation of q_∞ already defines the overall result, there is no need to hold on to a complete trace of the computation in u. We note that the resulting runtime and space savings can be significant and refer to [15] for a complete discussion of the non-accumulating WITH ITERATIVE.

Operating table u like a keyed dictionary (KEY, Figure 1c). Recall that recursive CTEs operate union table u in an append-only fashion (cf. assignment $u \leftarrow u \uplus i$ in Line 7 of Figure 1a). This changes with CTE variant KEY($k_1,\ldots,k_m$):

> When q_1 or q_∞ emit a row $t = (k_1,\ldots,k_m,c_1,\ldots,c_n)$, t replaces an older row of the same key $(k_1,\ldots,k_m)$ in table u. If there is no such row, t is simply added to u (upsert).

It is a runtime error if q_1 or q_∞ yield multiple rows sharing one key in any given iteration: see function $upsert(u,i)$ in Figure 2a which updates table u with the results i of the most recent iteration.

In effect, union table u behaves like a *keyed dictionary* (or *associative array*) which accepts updates of the form $u[(k_1,\ldots,k_m)] \leftarrow t$. Under the KEY variant, q_∞ may read the current state of the full dictionary via table name RECURRING(T) as well as access the rows added in the last iteration—the "hot rows" in the dictionary—as usual through T (note how tables w and u are passed to q_∞ in Line 5 of Figure 1c). The spread of key values, *i.e.*, the size of the active domain of columns $k_1,\ldots,k_m$, limits the size of table u and thus RECURRING(T), which queries can use to control space usage (see Section 3.1 for a quantitative assessment).

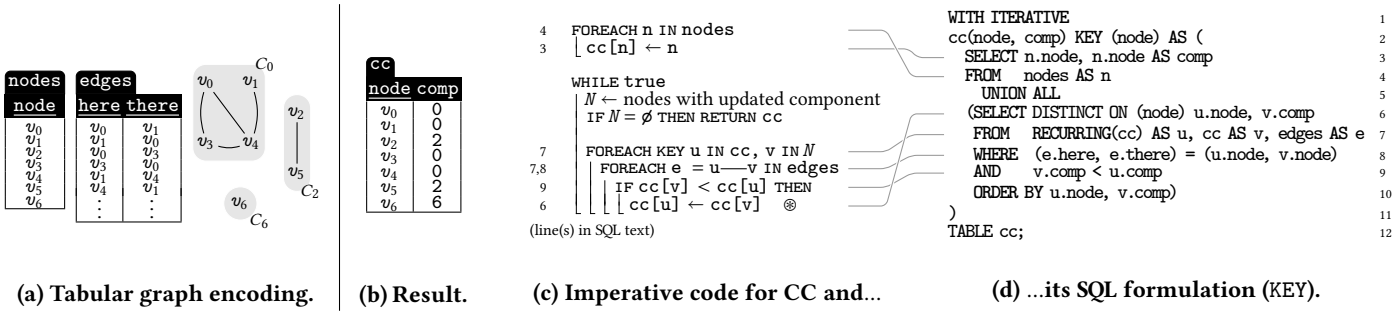
Most interestingly, access to the full dictionary and the ability to "overwrite" its entries, enable a query authoring style in which SQL code comes remarkably close to imperative (often stateful) formulations of iterative algorithms in which associative arrays are core data structures. Section 3.1 aims to demonstrate this, too.

Iteration with aging row memory (TTL, Figure 1d). CTE variant TTL(ttl) has been designed to address the effects of q_∞'s short-term row memory:

> When q_1 or q_∞ emit a row t with value $\ell \geqslant 0$ in column ttl, row t will remain accessible in table RECURRING(T)—the *recurring table* r—for the upcoming ℓ iterations.

As usual, T only holds those rows produced in the immediately preceding iteration—note how Line 5 in Figure 1d passes tables w

[1] In fact, if you cover up a non-monotonic q_∞ using shallow syntactic disguises, PostgreSQL evaluates the resulting WITH RECURSIVE with predictable and useful results.



(a) Tabular graph encoding. **(b) Result.** **(c) Imperative code for CC and...** **(d) ...its SQL formulation (KEY).**

Figure 3: CTE variant KEY admits the transcription of stateful iterative code into SQL (here: connected components).

and r to q_{∞}. Accordingly, reading the ttl value of a row in q_{∞} reveals its "*time (yet) to live*."

Since each row returned by q_{∞} may carry its individual ttl value, queries can ensure that table RECURRING(T) only contains those rows that are indeed relevant for the upcoming computation steps (applications of which are found in Sections 3.2 and 3.3). Keeping the cardinality of RECURRING(T) in check also aids efficient CTE evaluation. Emitting a row t with $t.ttl = 0$ submits t to the union table but does not let t recur in any future iteration, an idiom that queries can use to commit a result but then immediately forget about it (again, see Section 3.2 for a SQL query that employs this idiom).

The TTL extension is conservative in that it reduces to the vanilla WITH RECURSIVE if q_1 and q_{∞} (i) set column $ttl = 1$ for all emitted rows and otherwise (ii) do not read column ttl. Row expiry and aging are automatic and do not need to be expressed within q_{∞} itself (see Lines 2, 8, and 9 in Figure 1d): function $expire(\cdot)$ discards expired rows ($ttl = 0$) and then ages the remaining rows by decrementing their ttl value (Figure 2b).

PostgreSQL implementation. Since all new CTE variants directly derive their control and data flow from the vanilla WITH RECURSIVE, their prototypical implementation inside PostgreSQL v13 turned out to be straightforward (we put this to use in Section 3 below). To support variant KEY in particular, we could bank on PostgreSQL's own TupleHashTable and its associated support routines to implement $upsert(\cdot,\cdot)$ of Figure 2a. All database kernel changes remained local to PostgreSQL's original CTE execution code.

3 AN EXERCISE IN ITERATIVE CTEs

Formulating iterative queries in the absence of fixpoint-induced syntactic restrictions can be outright fun and lead to compact, sometimes even elegant, SQL formulations of a wide variety of algorithms. The following subsections aim to make this point and provide a taste of programming with the CTE variants of Section 2. We start out with an established problem over (tabular encodings of) graphs and, quite deliberately, end with one that pushes the database envelope (parsing based on context-free grammars), to provide an impression of what problems are comfortably in range.

3.1 Connected Components (KEY)

A keyed union table with upsert semantics can support the direct transcription of stateful iterative algorithms into SQL.

Here, we focus on finding the *connected components* in an undirected graph. An iterative CTE of variant KEY will operate over a graph encoding held in tables nodes and edges as shown in Figure 3a (note how an undirected edge u—v is encoded by two rows (u, v) and (v, u) in table edges). Two nodes share a component C if they are connected by any path: node v_0 thus shares component C_0 with v_1 while the unreachable v_6 sits in its separate component C_6. We are after a table cc (see Figure 3b) that assigns each node to its component (any unique identification of the components in column comp will do—here we reuse node IDs as component IDs).

Figure 3c depicts an imperative-style algorithm over tables nodes and edges that finds the graph's connected components. The procedure maintains an associative array cc in which an entry cc[v] = C indicates that node v is located in component C. This node-to-component assignment is iteratively updated (see the assignment marked by ⊛) until it becomes stable and each node has found its home component.

This algorithm design directly carries over to the SQL code shown in Figure 3d. The grey lines indicate where pieces of the imperative procedure find their place in the iterative CTE:

- In q_1, each node is initially assigned its own, unique component: the emitted row (n.node, n.node) corresponds with the assignment cc[n.node] ← n.node since column node has been declared key of table cc (see KEY(node) in Line 2).
- In q_{∞}, should node u be adjacent to v with a current component ID v.comp smaller than u's, assign u to v's component, too. In Line 6, q_{∞} thus emits row (u.node, v.comp), effectively performing the dictionary update cc[u.node] ← v.comp at ⊛.

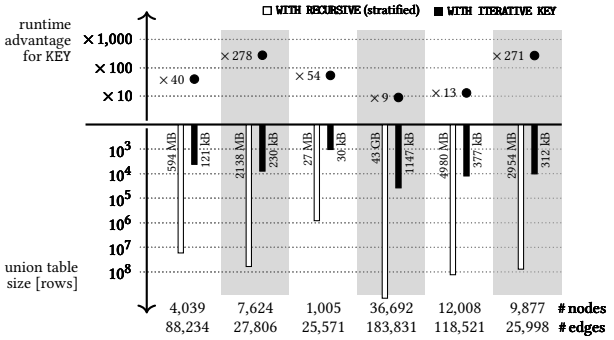


Figure 4: Running connected components on six graphs.

If node u has multiple neighbors $v_1, \ldots, v_m$, the minimum of their m component IDs will be u's component: in the imperative code, an earlier assignment to `cc[u]` will be overwritten by `cc[`v_i`]` should the latter be smaller (see the nested `FOREACH` loops in Figure 3c). The same effect is achieved by the pair `ORDER BY u.node, v.comp` and `DISTINCT ON (node)` on the SQL side which will pick the smallest `v.comp` ID for each `u.node`.

Note how the CTE can refer to table `RECURRING(cc)` to access the current node-to-component assignment for *all* nodes in the graph, while table `cc` only holds nodes whose assignment has most recently changed (cf. variable N on the imperative side). If no such changes are recorded in table `cc`, q_{cc} delivers zero rows, and the CTE will yield the then stable union table as the result (recall Figure 1c).

Cap on union table size. Due to `KEY(node)` and the upsert semantics, the union table will hold (at most) one entry per node—its size will thus *never exceed the cardinality of table* `nodes`.

This limit is notably lower than the union table size produced by a variant of *connected components* formulated using the original SQL:1999 fixpoint-based `WITH RECURSIVE`. This query will proceed in two phases (and thus is *stratified* [11]):

P1. Use the CTE to perform walks from all nodes u in the graph, returning row (u, v) if node v is reachable from u.
P2. For each u, the node v with minimum ID defines u's component.

Since P2 involves grouping (by u) and `MIN` aggregation, its computation cannot be folded into the SQL:1999 CTE of P1. This CTE will construct a, typically sizable, union table whose cardinality reflects the *number of non-intersecting paths* in the graph.

The impact of this union table size difference becomes tangible when we apply both, the `KEY` and stratified CTE variants, to a series of six graphs obtained from the SNAP archive [14], see the bottom half of the chart in Figure 4. While the `KEY` variant predictably constructs union tables holding one row per node, the stratified query and its CTE assemble union tables that may exceed 10^8 rows even for moderately sized graph instances. As is expected, PostgreSQL rewards the economical space usage of the `KEY`-based CTE with runtime reductions of factors from 10 to 100 and beyond (see the top half of Figure 4).

3.2 Twig Matching (`TTL`)

An iterative query benefits if it can be specific about the window of time in which prior result rows remain relevant. CTE variant `TTL` provides such "garbage row management" for table `RECURRING(`T`)`. This can aid query formulation and helps to reduce space usage as well as running time.

Here, we study this effect for the SQL query of Figure 5 that explores a dynamic search space of labeled nodes (or states). Initial query q_1 starts from the nodes `n` returned by *`start_nodes`*`()`, then q_{∞} iteratively expands the nodes `f` at the current fringe of the already visited search space using *`expand_node`*`(f.node)`.[2] In the resulting space of nodes, the SQL query aims to find node constellations of interest: the query of Figure 5 seeks the indicated three-layer twig

[2]We leave *`start_nodes`* and *`expand_node`* unspecified here. Either could be implemented in terms of, *e.g.*, a SQL UDF or a subquery. The successor nodes returned by *`expand_node`* may form an arbitrary DAG-shaped search space of finite size.

```
1  WITH ITERATIVE
2  fringe(ttl, node, tag, parent, match) TTL (ttl) AS (
3    SELECT 3 AS ttl,
4           n.node, n.tag, NULL AS parent, NULL AS match
5    FROM   start_nodes() AS n
6      UNION ALL -- recursive CTE
7    (SELECT 0 AS ttl,                                              match
8            b.node, b.tag, b.parent, array[a,b,c,d] AS match        a
9     FROM   RECURRING(fringe) AS a, RECURRING(fringe) AS b,         b
10           RECURRING(fringe) AS c, RECURRING(fringe) AS d        c   d
11    WHERE  (a.tag, b.tag, c.tag, d.tag) = ('a','b','c','d')
12    AND    (b.parent, c.parent, d.parent) = (a.node, b.node, b.node)
13      UNION ALL
14    SELECT DISTINCT ON (node, parent) 3 AS ttl,                    expand
15           n.node, n.tag, f.node AS parent, NULL AS match
16    FROM   fringe AS f, LATERAL expand_node(f.node) AS n
17   )
18 )
19 SELECT f.*
20 FROM   fringe AS f
21 WHERE  f.match IS NOT NULL;
```

Figure 5: Twig pattern matching (CTE variant `TTL`).

pattern of nodes a to d. We deliberately authored q_{∞} such that the *concerns of node expansion and twig matching are separated.*

Since we know that the twig pattern has depth three, nodes that are expanded by q_{∞} will become irrelevant after three further iterations of expansion (those future nodes will not be able to connect to branch root a). q_{∞} makes this observation explicit through column value `3 AS ttl` in its expansion part (see Line 14 and the `TTL(ttl)` clause in Line 2 of the query code). Table `RECURRING(fringe)` thus will only hold those nodes that are currently relevant for twig matching—older nodes are "forgotten" and will be inaccessible.

As a result, the matching part of q_{∞} (Lines 7 to 12) can opt for the most straightforward formulation of twig matching: row expiry ensures that the repeated self-joins over table `RECURRING(fringe)` do not risk an undesirable blow-up in join size. Note that matches are emitted using `0 AS ttl` (Line 7) which adds them to union table of all completed matches but will not let them recur in future iterations.

Layer-specific `TTL`. Still, q_{∞} can be even more specific about node aging and row garbage disposal. The iterated calls of *`expand_node`* explore the search space in layers: the nodes f_i at the fringe define the present, all nodes encountered in the same earlier iteration form a layer of the past (see Figure 6 in which layers are rendered the darker the longer they lie in the past). This layering applies to the twig pattern as well: nodes labelled a are located in the twig's root layer at height 3 seeking to connect (via b) to leaf nodes c or d two layers below. Query q_{∞} can express this label-based node-to-layer assignment once we replace `3 AS ttl` in Line 14 of Figure 5 by `CASE n.tag WHEN 'a' THEN 3 WHEN 'b' THEN 2 ELSE 1 END AS ttl`. The CTE's builtin row aging then ensures that nodes at layer 1 that do not occur in any twig—consider u and v in Figure 6, for

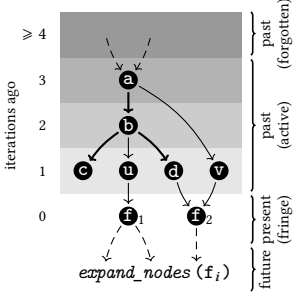


Figure 6: Layer-based aging.

example—will indeed expire after a single iteration (such nodes are only temporarily relevant to move the fringe forward).

The chart in Figure 7 reports that row aging at such finer granularity indeed further reduces the cardinality of table `RECURRING(fringe)` of non-expired rows. We conducted the experiment for search spaces between 2 and 4 million nodes but—as expected—this marginally impacts the average cardinality of the window of active nodes. Yet, since table `RECURRING(fringe)` is self-joined during twig matching, the query's run time increasingly benefits as we perform more matches with growing search space sizes.

3.3 CYK Parsing (TTL)

Quick row expiry helps iterative query efficiency. Holding on to rows *long enough*, however, can be required to ensure query correctness. Let us zoom in on this with one final TTL variant example which revolves around the Cocke-Younger-Kasami parsing algorithm (CYK) [21].

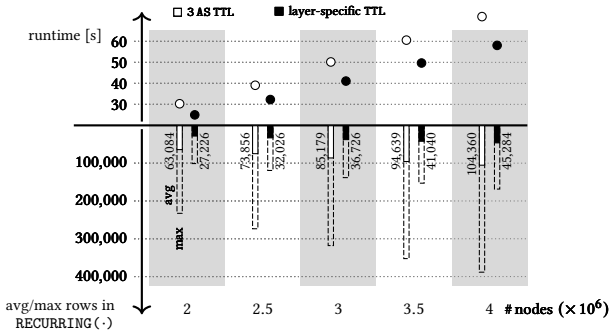


Figure 7: A specific TTL reduces the size of `RECURRING(·)`.

```
1  WITH ITERATIVE
2  parse(ttl, lhs, from, to) TTL (ttl) AS (
3    SELECT GREATEST(t.i-1, N-t.i) AS ttl, g.lhs, t.i AS from, t.i AS to
4    FROM   tokens AS t, grammar AS g
5    WHERE  t.sym ~ g.sym
6      UNION
7    SELECT GREATEST(l.from-1, N-r.to) AS ttl, g.lhs, l.from, r.to
8    FROM   RECURRING(parse) AS l, RECURRING(parse) AS r, grammar AS g
9    WHERE  l.to + 1 = r.from
10   AND    (g.rhs1, g.rhs2) = (l.lhs, r.lhs)
11 )
12 TABLE parse;
```

Figure 8: A formulation of CYK parsing in SQL (variant TTL).

CYK parses input token sequences based on context-free grammars in Chomsky normal form, in which the righthand side of a grammar rule either features one token or exactly two nonterminals [3]. Here, we use the input `6*(3+4)` and the grammar of parenthesized arithmetic expressions shown in Figure 9a to make CYK tangible. (Tokens need not represent characters, though, and other applications of parsing in a database context may involve complex feature extraction in time series data [18], for example.)

Ten lines of SQL (Figure 8) implement the complete CYK algorithm using a TTL CTE. The query operates over (i) table `tokens(sym,i)` in which the N rows (s, i) indicate that token s occurs at input position $1 \leqslant i \leqslant N$ (see the very bottom of Figure 9b), and (ii) table `grammar` (Figure 9a) in which each row encodes one rule of the Chomsky grammar.

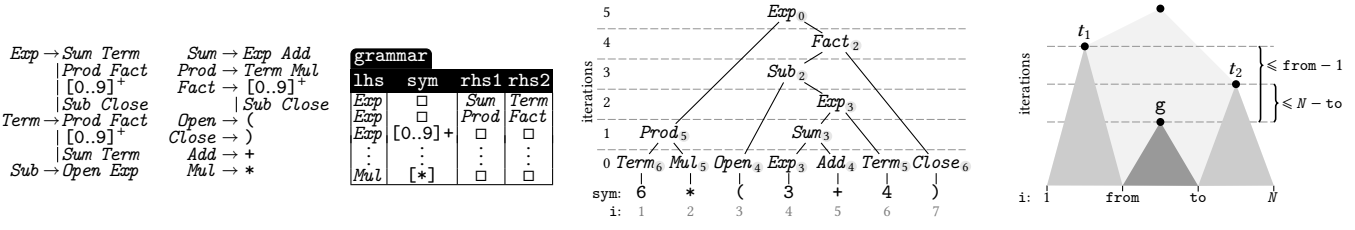
CYK—and thus the CTE—proceed bottom-up: q_1 finds (the lefthand sides `lhs` of) the rules that produce the individual tokens in the input sequence. In subsequent iterations, q_{∞} combines (or, literally, joins) two token sequences `l` and `r`, provided that they are adjacent in the input (`l.to+1 = l.from`) and that a rule `g` = `g.lhs` → `g.rhs1 g.rhs2` in `grammar` derives their lefthand sides: `(g.rhs1,g.rhs2) = (l.lhs,r.lhs)`.

(Aside: We operate this CTE in set-based `UNION` semantics to avoid to learn about identical parses multiple times. As for SQL:1999 CTEs, the use of `UNION` indicates that no row t will find its way into intermediate table i if t is already in union table u. Essentially, in the semantics of Figure 1d, replace Line 5 by $i \leftarrow q_{\infty}(w, r) \setminus u$.)

Remembering parses as long as needed... Both, `l` and `r`, are found through two lookups in `RECURRING(parse)`, the table of already known partial parses. At this point, it *does not suffice* to refer to the partial parses found in the immediately preceding iteration (see the layered parse tree in Figure 9b): when non-terminal *Sub* is to be established in iteration 3, its left partial parse *Open* had already been found by q_1 in iteration 0, *i.e.*, three iterations ago. Likewise, *Exp* in iteration 5 can only be established if *Prod* from iteration 1 still recurs as a relevant partial parse.

...**but no longer than that.** Do we thus hold on to partial parses for the entire parsing process, *i.e.* for the maximum of $N - 1$ iterations required to build a left-deep (or right-deep) parse tree for the input of length N? That would be safe but wasteful. Much like in Section 3.2, we can use algorithmic insight to let the query dispose of rows early.

Refer to Figure 9c. Since the current rule `g` derives the input tokens from position `l.from` to `r.to`, the parse trees t_1 and t_2



(a) Expression grammar in Chomsky normal form. **(b) Layered parse tree for `6*(3+4)`.** **(c) Parse tree height limits.**

Figure 9: A TTL CTE can use deliberate assignments of *time to live* values (see annotations in) to expire partial parses early.

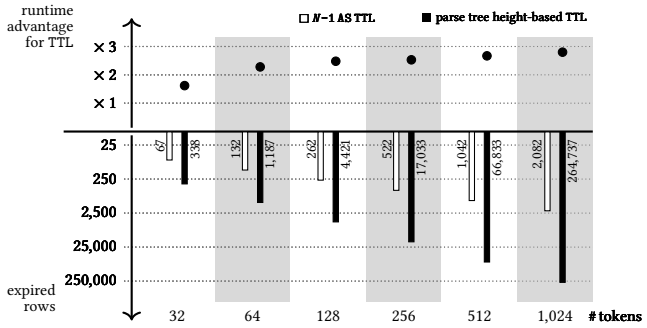


Figure 10: SQL-based CYK parser: runtime and row expiry.

to the immediate left and right will derive token sequences of length `l.from-1` and *N*`-r.to`, respectively. Even if these parse trees take the shape of left- or right-deep linear chains, it will take no more than `GREATEST(l.from-1,`*N*`-r.to)` iterations to construct them (see the computation of column `ttl` in Line 7 of Figure 8—the ● in Figure 9b annotate the parse tree nodes with their resulting `ttl` value). If we keep `g`'s partial parse for the indicated number of iterations, it is guaranteed that its parse tree can be joined with either t_1 or t_2. Beyond that point, we can let the partial parse expire from `RECURRING(parse)`.

Compared to the conservative *time to live* *N*`-1`, this simple change to the query indeed leads to significantly faster row expiry (cf. the bar chart in the bottom half of Figure 10, we measured the white bars when we replaced the underlined expressions in Figure 8 by *N*`-1 AS ttl`).

Non-linear recursion: `TTL` vs. vanilla `WITH RECURSIVE`. A two-fold reference to earlier partial parses leads to the elegant SQL formulation of CYK in Figure 8. In absence of `TTL` or `KEY` CTEs, yet again we would need to work around SQL:1999 restrictions to allow multiple working table references. On top of the syntactic tricks we would be required to play in PostgreSQL (cf. Section 2), in each iteration it would be the query's own responsibility to explicitly add the current working table contents to q_{∞}'s result. Only then do we obtain the expected behavior of non-linear recursion [2] and implement the longer-term row memory required by CYK. We are definitely entering the territory of syntactic atrocities that was already criticized in Section 1.

While this hurts the query's readability, it also impacts its performance. When we run the CYK parser against token sequence of growing lengths, we find the expected runtime advantage of a proper in-kernel `TTL` implementation over this tinkering with non-linear recursion in vanilla PostgreSQL (see the upper half of Figure 10).

4 EARLIER AND FUTURE WORK

Regarding semantics, `WITH RECURSIVE` may be the odd man out among the SQL language constructs. The CTE's unique ability to express arbitrary iterative in-database computation, however, makes it an essential building block if complex algorithms are to be evaluated close to tabular data. Efforts that address the applicability and efficiency of recursive CTEs thus abound.

In *RaSQL*, CTEs retain their fixpoint semantics, yet specific forms of aggregation and grouping are admissable if q_{∞} exhibits the *PreM* property [11, 22]. This enables, for example, a formulation of *connected components* that resembles the code in Figure 3d. Much like we observed in Section 3.1, in the absence of stratification, *RaSQL*'s Spark-based implementation can improve the running time of *connected components* by a factor of 100. The recent *Datalog*° effort [13], too, addresses the interleaving of recursion and aggregation, focusing on the optimization of the actual looping logic (as opposed to the loop's body).

A variation of the `WITH ITERATIVE KEY` semantics (recall Figures 1b and 1c) led the *DBSpinner* project [9] to a new iterative query construct that aids query readability. We share their observation that the in-kernel implementation of such constructs is material to efficient evaluation. *SQLoop* [10] follows a different path and iteratively drives the evaluation of CTE variants from outside the RDBMS.

Unlike PostgreSQL, *MariaDB* [19] offers a configuration option that admits non-linear recursion in CTEs. Working table w then holds *all* rows found in earlier iterations—instead, we propose to give queries fine-grained control over row retention and expiry in table `RECURRING(`*T*`)`.

Beyond `KEY` and `TTL`. Behind the scenes of PL/SQL translation [6], CTEs continue to be a great compilation target. Still, we are underway to further develop the pragmatics and efficiency of CTEs when they are used as user-facing SQL constructs. CTE variants currently on our workbench include:

- Iterative queries that may place an intermediate result row t into one of *multiple different working tables* (selected by the value in designated column $t.wt$, much like we introduced column ttl).
- Modifiers (like `RECURRING(·)`) that lead the CTE to maintain rows in a special *working stack* w (as opposed to a table) such that successive iterations of q_{∞} can read earlier rows using a LIFO discipline.

On the side of in-kernel underpinnings, the latter CTE variant—just like `KEY` and `TTL`—will benefit from dedicated internal representations of w (*e.g.*, in terms of a ttl-based priority queue in the case of `TTL` to speed up expiry). This, too, is currently in the works.

REFERENCES

[1] F. Bancilhon. Naive Evaluation of Recursively Defined Relations. In *On Knowledge Base Management Systems*, pages 165–178. Springer, 1986.
[2] F. Bancilhon and R. Ramakrishnan. An Amateur's Introduction to Recursive Query Processing Strategies. *ACM SIGMOD Record*, 15(2):16–52, June 1986.
[3] N. Chomsky. On Certain Formal Properties of Grammars. *Information & Computation*, 2(2):137–167, 1959.
[4] C. Duta. Another Way to Implement Complex Computations: Functional-Style SQL UDFs, June 2022.
[5] C. Duta and T. Grust. Functional-Style SQL UDFs with a Capital 'F'. In *Proc. SIGMOD*, Portland, OR, USA, June 2020.
[6] C. Duta, D. Hirn, and T. Grust. Compiling PL/SQL Away. In *Proc. CIDR*, Amsterdam, The Netherlands, January 2020.
[7] A. Eisenberg and J. Melton. SQL:1999, Formerly Known as SQL3. *ACM SIGMOD Record*, 28(1):131–138, March 1999.
[8] S.J. Finkelstein, N. Mattos, I. Mumick, and H. Pirahesh. Expressive Recursive Queries in SQL. Joint Technical Committee ISO/IEC JTC 1/SC 21 WG 3, Document X3H2-96-075r1, March 1996.
[9] S. Floratos, A. Ghazal, J. Sun, J. Chen, and X. Zhang. DBSpinner: Making a Case for Iterative Processing in Databases. In *Proc. ICDE*, Chania, Greece, April 2021.
[10] S. Floratos, Y. Zhang, Y. Yuan, R. Lee, and X. Zhang. SQLoop: High Performance Iterative Processing in data Management. In *Proc. ICDCS*, Vienna, Austria, July 2018.

[11] J. Gu, Y.H. Watanabe, W.A. Mazza, A. Shkapsky, M. Yang, L. Ding, and C. Zaniolo. RaSQL: Greater Power and Performance for Big Data Analytics with Recursive-Aggregate-SQL on Spark. In *Proc. SIGMOD*, Amsterdam, The Netherlands, June 2019.

[12] D. Hirn and T. Grust. One `WITH RECURSIVE` is Worth Many `GOTO`s. In *Proc. SIGMOD*, Xi'an, Shaanxi, China, June 2021.

[13] M.A. Khamis, H.Q. Ngo, R. Pichler, D. Suciu, and Y.R. Wang. Datalog in Wonderland. *ACM SIGMOD Record*, 51(2):6–17, June 2022.

[14] J. Leskovec and A. Krevl. SNAP Datasets: Stanford large network dataset collection. `http://snap.stanford.edu/data`, June 2014.

[15] L. Passing, M. Then, N. Hubig, H. Lang, M. Schreier, S. Günnemann, A. Kemper, and T. Neumann. SQL- and Operator-Centric Data Analytics in Relational Main-Memory Databases. In *Proc. EDBT*, Venice, Italy, March 2017.

[16] *PostgreSQL (version 13) Documentation*. `http://www.postgresql.org/docs/13/`.

[17] *PostgreSQL (version 14) Documentation*. `http://www.postgresql.org/docs/14/`.

[18] P. Senin, J. Lin, X. Wang, T. Oates, S. Gandhi, A.P. Boedihardjo, C. Chen, and S. Frankenstein. GrammarViz 3.0: Interactive Discovery of Variable-Length Time Series Patterns. *ACM TKDD*, 12(1):1–28, February 2018.

[19] G. Shalygina and B. Novikov. Implementing Common Table Expressions for MariaDB. In *Proc. SEIM*, St. Petersburg, Russia, April 2017.

[20] *SQL:1999 Standard. Database Languages–SQL–Part 2: Foundation*. ISO/IEC 9075-2:1999.

[21] D.H. Younger. Recognition and Parsing of Context-Free Languages in Time n^3. *Information and Control*, 10:189–208, 1967.

[22] C. Zaniolo, M. Yang, A. Das, A. Shkapksy, T. Condie, and M. Interlandi. Fixpoint Semantics and Optimization of Recursive Datalog Programs with Aggregates. *Theory and Practice of Logic Programming*, 17(5–6):1048–1065, September 2017.